Project 3: Distance Vector Routing Protocol
(CS3516 – B16)

Due: Dec 13, 2016, 11:59 PM

Total Points
100 (One Hundred)

1. Overview
In this assignment you will be asked to implement a distributed asynchronous distance vector (DV) routing protocol. This project will be completed using C/C++ over a network emulator already provided. No knowledge of Unix system calls will be needed to complete this project.

In this programming assignment, you will be writing the sending and receiving IP-level messages for routing data over a network

Your code must execute in an emulated hardware/software environment. The programming interface provided to your routines, i.e., the code that would call your entities from above and from below is very close to what is done in an actual Linux environment.

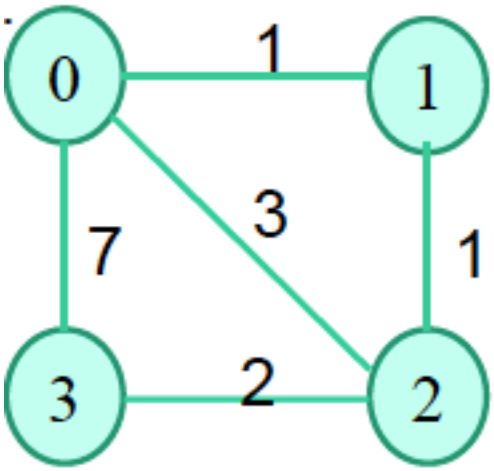

Figure 1: Default Network Configuration

2. Routines
In this project, you will be writing a "distributed" set of procedures that implement a distributed asynchronous distance vector routing for a network. Figure 1 shows the default node configuration that you will use to develop and test your code.

IMPORTANT NOTE: You code should work for the default node configuration in Figure 1. In addition, we will test your code with a secret node configuration (with 4 nodes). This means the links between the nodes may change and so might the weights compared to Figure 1. Your code should be general enough to work on any configuration of 4 nodes.

The rest of the document is described primarily in terms of the default node configuration given in Figure 1.

For this assignment, you are to write the following routines, which will ``execute" asynchronously within the emulated environment that we have written for this assignment.

These routines exist in **node0.c**, and similarly named routines live in each of the **modeN.c** files.

rtinit0() This routine will be called once at the beginning of the emulation**. rtinit0()** has no arguments. It should initialize the distance table in **node0** to reflect the direct costs to is neighbors by using **GetNeighborCosts().** In Figure 1, representing the default configuration, 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 **node0** routines, it should then send to its directly-connected neighbors (in the Figure, 1, 2 and 3) 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 routine **tolayer2(),** as described below. The format of the routing packet is also described below.

rtupdate0(struct rtpkt *rcvdpkt). This routine will be called when node 0 receives a routing packet that was sent to it by one if its directly connected neighbors. The parameter ***rcvdpkt** is a pointer to the packet that was received **rtupdate0()** 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. **rtupdate0()** 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, for the example in Figure 1, **nodes 1 and 2** will communicate with each other, but **nodes 1 and 3** will not communicate with each other. As we saw/will see in class, 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 N-by-N array of int's, 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 9999 is ``infinity.''

Figure 2 provides a conceptual view of the relationship of the procedures inside node 0.
Similar routines are defined for nodes 1, 2 and 3. Thus, you will write, at a minimum, 8 procedures in all: **rtinit0(), rtinit1(), rtinit2(), rtinit3(),rtupdate0(), rtupdate1(), rtupdate2(), rtupdate3().**

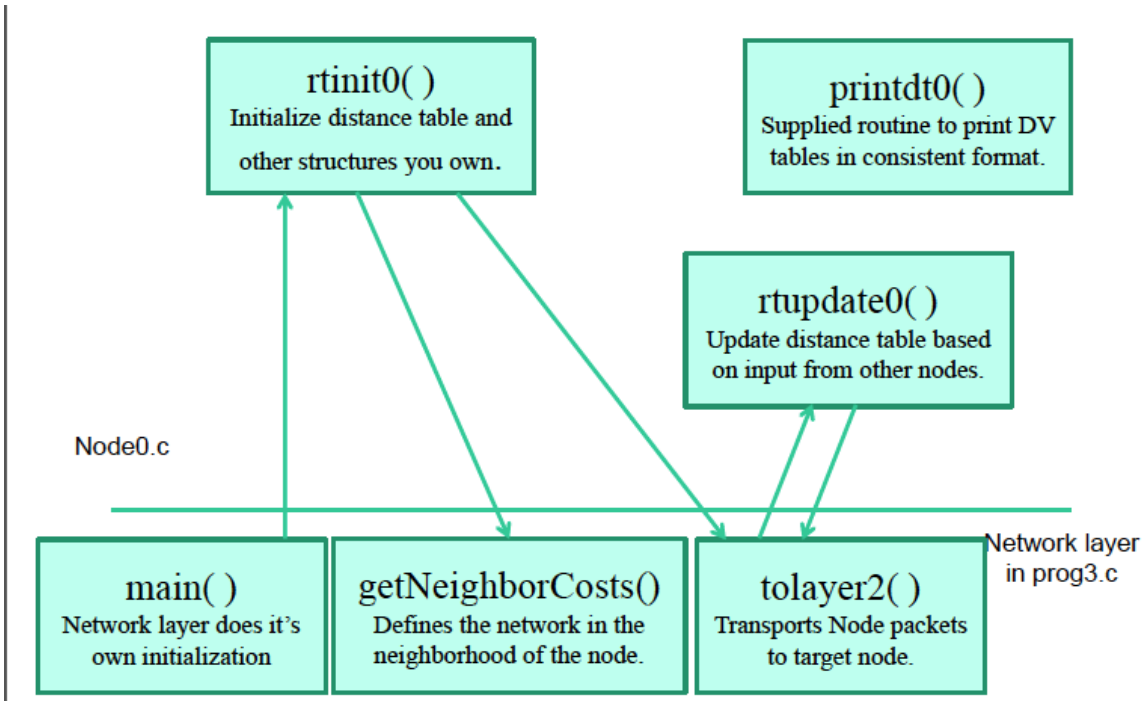

Figure 2: Node Procedure Relationships

3. Software Interfaces

The procedures described above are the ones that you will write. We have written the following routines that can be called by your routines:

`toLayer2(struct rtpkt pkt2send);` where **`struct rtpkt`** is defined below. The purpose of this routine is to provide communication between the various nodes in your network. The procedure **`tolayer2()`** is defined in the file **`project3.c`**

`getNeighborCosts( int myNodeID );` This routine allows your node to discover what other nodes are around it and what are the costs to those nodes. This routine returns a pointer to a struct NeighborCosts that is defined on a later slide. **`getNeighborCosts()`** is implemented in **`project3.c`**

`printdt0( int MyNodeNumber, struct NeighborCosts *neighbor, struct distance_table *dtptr );`

will pretty print the distance table for **node0**. Please look in your **`node0.c`** code for an explanation of how it works and what you need in order to call it. What should be of special interest to you is that this print code is the same in each of the nodes; in other words, I'm using the input parameters to tailor the routine to work in general for all nodes. There is no need for me to write a separate and unique routine for each node.

4. Software Structures

These are the data structures that you will use to communicate with the routines in Layer 2. You will certainly need additional structures of your own crafting to maintain additional information.

```
extern struct rtpkt {
int sourceid; // id of node sending this packet, 0, 1, 2, or 3
int destid; // id of router to which pkt is sent
int mincost[MAX_NODES]; // min cost to node 0 ... N
};
```

An instance of this structure is declared in your starter node0.c … node3.c code.

```
struct NeighborCosts {
int NodesInNetwork; // The total number of nodes in the entire network
int NodeCosts[MAX_NODES]; // An array of
};
```

An instance of a pointer to this structure is declared in your starter **node0.c … node3.c** code. Note that the filled-in structure is allocated in the function **getNeighborCosts()** and a pointer to this structure is returned by this function. Here's the information returned as seen by **gdb** in my **node0.c** when using the default configuration shown in Figure 1.

```
(gdb) p *neighbor0
$2 = {NodesInNetwork = 4, NodeCosts = {0, 1, 3, 7, 9999, 9999, 9999,
9999, 9999, 9999}}
```

MAX_NODES is set to 10. The simulator is able to support a network containing up to 10 nodes. **But you will not need to work with more than 4 nodes.**

6. Environment

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.

Your project will be evaluated on the CCC machines.

To compile your routines:

```
gcc project3.c node0.c node1.c node2.c node3.c -o project3
```

You can download the code from here:
http://hibou.cs.wpi.edu/~kven/courses/CS3516-B16/projects/Project3.zip

7. Output Trace

For your sample output, your procedures should print out a message whenever your **rtinit0(), rtinit1(), rtinit2(), rtinit3()** or **rtupdate0(), rtupdate1(), rtupdate2(), rtupdate3()** procedures are called, giving the time (available via my global variable clocktime).

For **rtupdate0(), rtupdate1(), rtupdate2(), rtupdate3()** 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, print the contents of the distance table (you should use my pretty-print routines),
- A description of any messages sent to neighboring nodes as a result of any distance table updates.

We will be evaluating your output listing with a TraceLevel value < 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 our emulator will terminate. Bellow you can see a sample output.

```
G:\Courses\3516\Project3\solution>p3 1
At time t=0.000, rtinit0() called.
```

```
via
D0 | 1 2 3
----|-----------------------------
dest 1| 1 9999 9999
dest 2| 9999 3 9999
dest 3| 9999 9999 7
At time t=0.000, node 0 sends packet to node 1 with: 0 1 3 7
At time t=0.000, node 0 sends packet to node 2 with: 0 1 3 7
At time t=0.000, node 0 sends packet to node 3 with: 0 1 3 7
At time t=0.000, rtinit1() called.
via
D1 | 0 2
----|-----------------------------
dest 0| 1 9999
dest 2| 9999 1
dest 3| 9999 9999
At time t=0.000, node 1 sends packet to node 0 with: 1 0 1
9999
At time t=0.000, node 1 sends packet to node 2 with: 1 0 1
9999
```

10. FINAL NOTES

- **The format of the output of the DV tables in this assignment is different from the one in the book. So be careful!**
- **If your best route from one node to another ends up having a loop, that's okay!**

11. Submission

As a part of the submission you need to provide the following:

Create a zip file **"your-wpi-login_project3.zip"**

1. All the code for the distance vector (DV) protocol (any .c/.c++ and .h files), including any code that was pre-provided as part of this project
2. A ReadMe (txt) file that spells out exactly how to compile and run the code
3. Makefile to make compilation easy.

Submit your document electronically via Instruct Assist (https://ia.wpi.edu/cs3516/) by 11:59pm on the day the assignment is due. Make sure you choose "Project 3" under project drop-down in IA before uploading the zip file.

12. Grading

Grade breakup (%) assuming all documents are present:

1. Functioning DV Protocol code with default node configuration given in Figure 1 = 45%
2. Functioning DV Protocol code with our secret 4 node configuration = 45%
3. README = 5%
4. Makefile = 5%

The grade will be a ZERO if:

1. If the code does not compile or gives a run-time error
2. If README with detailed instructions on compiling the running the code is not present