Computer Networks

Project 2

Implementing a Routing Protocol

Due: December 11 until 23:59

Place to submit : C709

Implementing a Routing Protocol

Description

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.

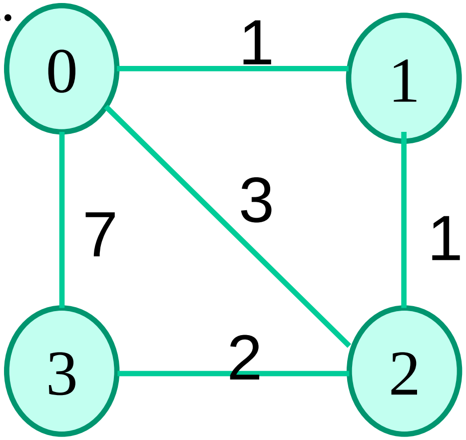
Details

1. **A detailed project description can be found on later slides.**
2. In this programming assignment, you will be writing the sending and receiving IP-level messages for routing data over a network
3. 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.

Implementing a Routing Protocol

Overview

In this lab, you will be writing a "distributed" set of procedures that implement a distributed asynchronous distance vector routing for the network shown in **Figure 1.**



The Basic Assignment

The routines you will write: 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 nodeN.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 node 0 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 node 0 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.

Implementing a Routing Protocol

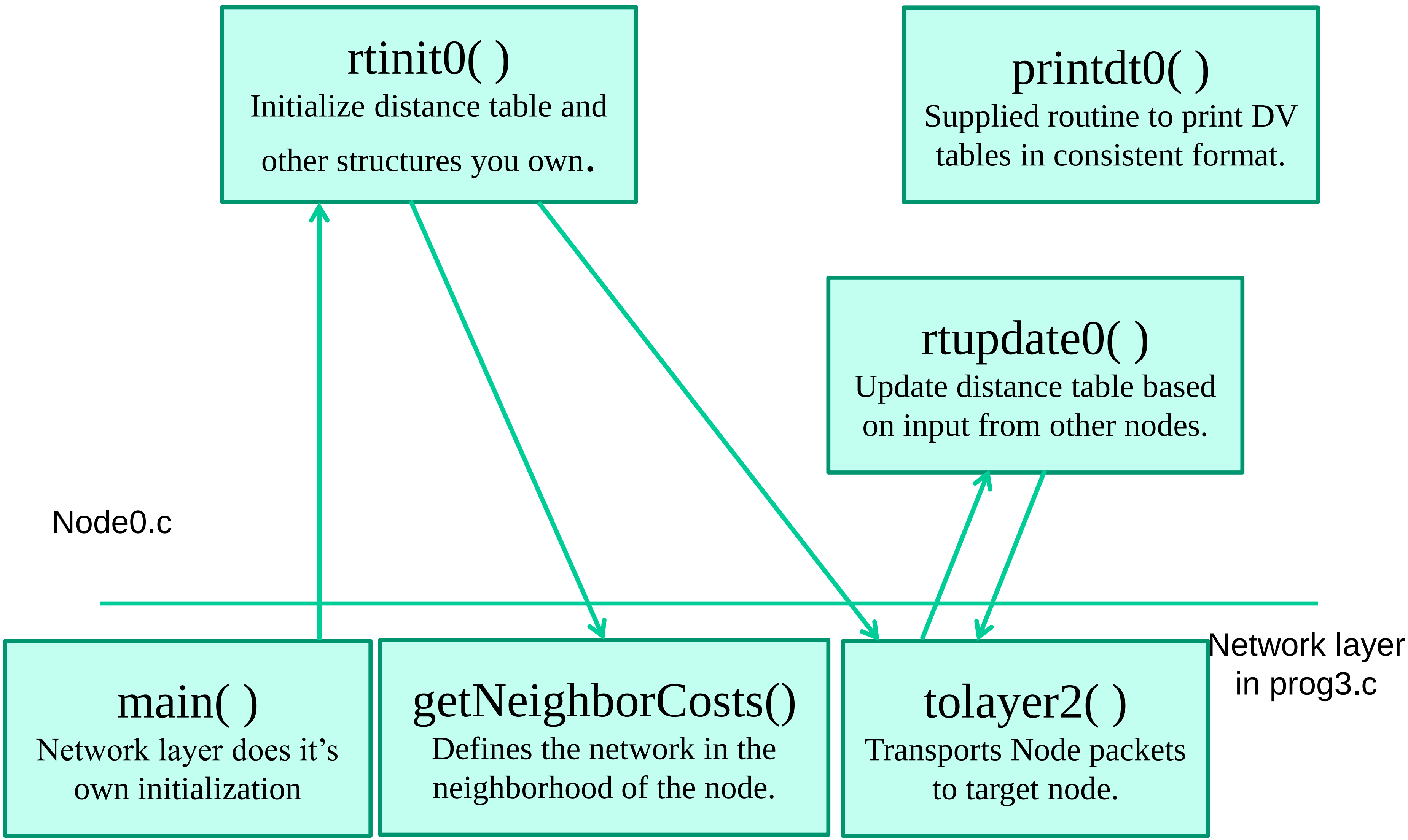
rtupdate0() is the "heart" of the distance vector algorithm. The values it receives in a routing packet from some other node *i* contain *i*'s current shortest path costs to all other network nodes. rtupdate0() uses these received values to update its own distance table (as specified by the distance 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 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()`

Implementing a Routing Protocol



Implementing a Routing Protocol

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 node 0. 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.

Implementing a Routing Protocol

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.

Implementing a Routing Protocol

The Simulated Network Environment

Your procedures rtinit0(), rtinit1(), rtinit2(), rtinit3() and 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 sender and receiver is variable (and unknown).

When you compile your procedures and my procedures together and run the resulting program, you will be asked to specify only one value regarding the simulated network environment:

- **Tracing.** Setting a tracing value of 1 or greater 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 greater than 2 will display all sorts of odd messages that are for my own emulator-debugging purposes.

A tracing value of 2 is reserved for your use in your code. We will typically be running Show and Tell with a trace level of 1, so the output will not be encumbered with your output at that time. In the initial node0.c code, *TraceLevel* is given to you as an external.

You can set the trace level as the only input on the command line: "`Program3 1`".

The Assignment

You are to write the procedures rtinit0(), . . . rtinitN() and rtupdate0(), . . . rtupdate3(), which together will implement a distributed, asynchronous computation of the distance tables for the topology and costs shown in Figure 1.

Implementing a Routing Protocol

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. To compile your routines:

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

This assignment can be completed on any machine supporting C

Manifest:

Here are the files you will need for this project:

[project3.c]

[project3.h]

[node0.c]

[node1.c]

[node2.c]

[node3.c]

[NodeConfigurationFile]

Implementing a Routing Protocol

Output Trace (for submission): 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

(1) Print the identity of the sender of the routing packet that is being passed to your routine,

(2) Whether or not the distance table is updated, print the contents of the distance table (you should use my pretty-print routines),

(3) A description of any messages sent to neighboring nodes as a result of any distance table updates.

The submitted output will be an 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.

Implementing a Routing Protocol

Here's a sample output produced with my solution of the problem I believe it follows the guidelines on the previous page. It is not the intent that you should slavishly follow this character for character.

```
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
```

Implementing a Routing Protocol

Project Evaluation

Project Object

Source Code(Detail Comment)

Result screen

Result analysis and discussion

Reference