Computer Networks Programming Assignment 2

Fall 2023

Due: December 4, 2023, 11:59 PM

This project assignment is based on the Programming Assignment of Chapter 5 of our textbook, in which you will be writing a "distributed" set of procedures that implements a distributed asynchronous distance-vector (DV) routing algorithm we just learned. The topology of the network is shown in Figure 1. You are recommended to use C/C++ to do the project, and a network emulator (in C) is provided to help you. If you strongly prefer to use Java or Python, check out the guideline provided in Sec. 3. No knowledge of Unix system calls will be needed to complete this project.

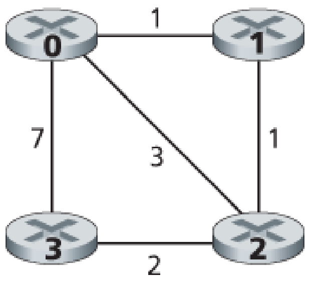


Figure 1: The topology for the network

1 Requirements

The C template accompanied with this project contains 5 major C source files: node0.c, node1.c, node2.c, node3.c, dv.c, which are corresponding to node 0, 1, 2, 3 in Figure 1 and the network simulator. The C source files contain a few important routines for the project (but they are empty now):
node0.c: rtinit0(), rtupdate0();
node1.c: rtinit1(), rtupdate1();
node2.c: rtinit2(), rtupdate2();

node3.c: rtinit3(), rtupdate3().
Your task for the project is to implement routines rtinit0(), ..., rtinit3() and rtupdate0(), ..., rtupdate3(), such that they can work together to implement a distributed, asynchronous computation of the distance tables for the topology and costs corresponding to Figure 1. 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 all the C source files, you can use:

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

A Little More Details on Routines

We use the routines in node 0 as examples. The routines in other nodes are all similar.

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 (node 1, 2 and 3 in Figure 1) 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.

void rtupdate0(struct RoutePacket *rcvdpkt): This routine will be called when node 0 receives a routing packet that was sent to it by one of 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 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 an N-by-N matrix 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.". We have provided the default configuration file (i.e., NodeConfigurationFile) corresponding to Figure 1, and the organization of the configuration can be found in Appendix A. For the basic assignment (non extra credit), you should not need to modify the provided configuration file.

Figure 2 provides a conceptual view of the relationship between the procedures of node 0 (node0.c) and the simulator (dv.c). Similar routines are defined for nodes 1, 2 and 3.

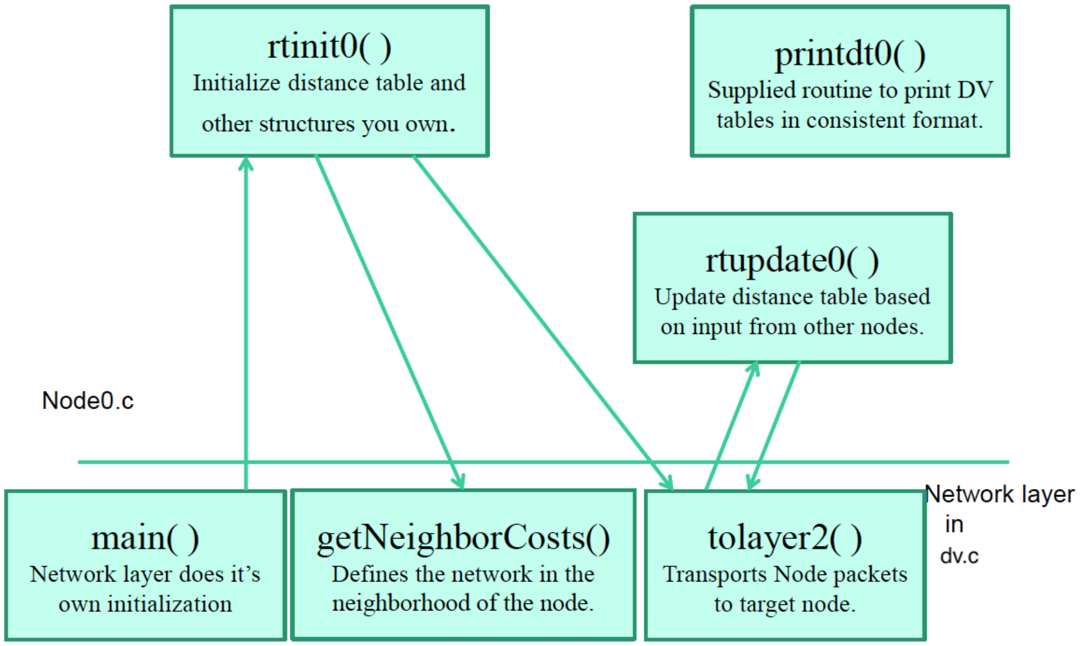

rtinit0( )
Initialize distance table and other structures you own.
printdt0( )
Supplied routine to print DV tables in consistent format.
rtupdate0( )
Update distance table based on input from other nodes.
Node0.c
Network layer in dv.c
main( )
Network layer does it's own initialization
getNeighborCosts()
Defines the network in the neighborhood of the node.
tolayer2( )
Transports Node packets to target node.

Figure 2: Relationship between the procedures of node 0 and the simulator

Software Interfaces

We have provided the following routines in dv.c that can be called by your routines:

toLayer2(struct RoutePacket packet): The purpose of this routine is to provide communication between the various nodes in your network. The struct RoutePacket is defined below.

getNeighborCosts(int myNodeNumber): 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.

printdt0(int MyNodeNumber, struct NeighborCosts *neighbor, struct distance_table

***dtptr)**: This routine 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 similar in each of the nodes.

Data 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.

```
struct RoutePacket {
int sourceid; // id of sending router sending this pkt
int destid; // id of router to which pkt being sent
// (must be an immediate neighbor)
int mincost[MAX_NODE];// min cost from node 0 ... N
};
```

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

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.

The Simulated Network Environment

The simulator is implemented in dv.c (this should remain unchanged for your project, I believe). Your procedures rtinit0(), rtinit1(), rtinit2(), rtinit3() and rtupdate0(), rtupdate1(), rtupdate2(), rtupdate3() in node0.c, node1.c, node2.c, and node3.c 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 routines and the provided simulator together and run the resulting program (e.g., ./dv), you will be asked to specify a "Trace Level" (you can also enter the "Trace Level" using a command line argument, e.g., ./dv 1). The "Trace Level" is defined as follows:

Setting a tracing value (defined by the variable "*TraceLevel*" in the source code) 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 own use in your code. The grader will be running tracing value 1 only, so the output will not be encumbered with your personal output at that time. Therefore, make sure the program will print the necessary information for grading when the trace value is 1. In the initial node0.c, node1.c, node2.c, node3.c code, *TraceLevel* is given to you as an external.

Output Trace

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 the global variable *clocktime*).

For rtinit0(), rtinit1(), rtinit2(), rtinit3(), you should:
1) Print the content of the initial distance table (e.g., use print routine available in the template), and
2) Print what information is sent to the neighboring nodes by this node at what time initially (see Figure 3).

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, and
2) Whether or not the distance table is updated, print the content of the distance table (e.g., use print routine available in the template), and
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 = 1. 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 (the routing protocol has converged), at which point our emulator will terminate.

Sample output. Some example output is shown in Figure 3 (from Prof. Jerry Breecher's CS3516-Computer Networks).

2 Collaboration Policy

You are allowed to work on this project in a group of at most 2 students from CS4461/EE4272. 3 or 3 more students in a group will be not acceptable and will result in 0 grade for each. You can also work on the project individually. However, there will be no bonus points for individual work.

```
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
```

Figure 3: Some example output

3 Deliverables

You are recommended to write the programs in C/C++ (i.e., *directly modifying the provided C template*). Note that this version of C template was originally from the textbook (`https://media.pearsoncmg.com/aw/aw_kurose_network_3/labs/lab6/lab6.html`) and has been revised by Jerry Breecher. However, if you want, feel free to start from the original version of the C template which is available in `https://media.pearsoncmg.com/aw/aw_kurose_network_3/labs/lab6/lab6.html`). If you want to write the programs in Java, you may check out the Java template provided by our textbook in `https://media.pearsoncmg.com/aw/aw_kurose_network_3/labs/lab6/lab6.html` (I have not checked their Java template, and you need to double check whether it works or not). If you want to write the program in Python, no template will be available and you need to create everything from scratch following the project description (including the network simulator). This would be challenging and I don't recommend you to use python.

Regardless what programming language you will be using, it is your responsibility to prepare a README file to tell the grader how to deploy, compile and run your code. If the code cannot be compiled and run, you will not get any points and therefore, make sure someone can follow your README file to deploy, compile (*a makefile or a script for compilation should be provided if needed*) and run your code and, if any additional packages are needed, they should come together with your submission and instructions on how to use those packages should be included in your README file. Also, if you are working in a group, please also include a COLLABORATORS file, which contains the information about students involved in this project, including first name, family name, and the student ID for each. Up to 2 students are allowed in a group, and only one group member needs to submit the project outcome (i.e., our late policy will be enforced on the student who submits the project outcome).

It is preferred that all files be zipped up and submitted as a single package. Tarballs, Gzipped and "plain" Zip files are great. The packet should be submitted to Canvas by one group member.

4 Grading

Grade breakdown (out of 100 points):

- DV routing protocol code functioning correctly on the default supplied configuration (4 nodes as described in NodeConfigurationFile) = **60**.
- README/Makefile describing how to compile and run your code = **10**
- Design document for the corresponding DV routing protocol (i.e., theoretic process including initial phase and the convergence phase) = **20**. This means, without writing

any program, and if you can submit the design document with correct results, you will receive 20 points

- Output traces are correctly formatted according to the project description = **10**

The grade will be a **0**:

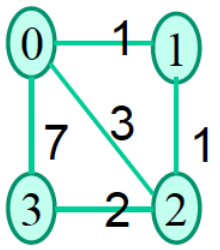
- If the code does not compile or gives a run-time error.
- If README with detailed instructions on compiling/ running the code is not present.

5 Extra Credit

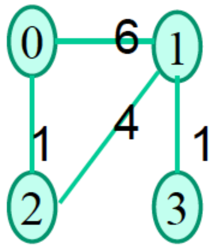
For 10 extra points, your program can support up to 10 nodes (as defined in MAX_NODES in dv.h). To do the extra-credit assignment, you need to understand how the NodeConfigurationFile is organized (you don't need to know this for the basic assignment, I believe). Check out Appendix A to understand the organization of NodeConfigurationFile.

A The Organization of NodeConfigurationFile

The NodeConfigurationFile is organized using a simple distance matrix. Element [i,j] in the matrix means the cost (distance) from node i to node j and, if node i and j is un-reachable, the cost is 9999 (equivalent to infinity in our lecture). Please see Figure 4 for some sample configurations.



```
This is the original configuration:
4
   0,    1,    3,    7
   1,    0,    1, 9999
   3,    1,    0,    2
   7, 9999,    2,    0
```



```
The file for this configuration:
4
   0,    6,    1, 9999
   6,    0,    4,    1
   1,    4,    0, 9999
9999,    1, 9999,    0
```

Figure 4: Sample configurations of NodeConfigurationFile