ECE 463 - Introduction to Computer Networks Fall 2017

Project II: Distance Vector Routing

Checkpoint Due	11/03/2017, Friday, 11:59 AM (Noon)
Due Date	11/17/2017, Friday, 11:59 AM (Noon)

To be done individually or in groups of two

Overview:

The goal of this project is to implement a simplified version of the distance vector protocol, and explore key issues in the design of a distance vector protocol.

Distance Vector Routing:

In a Distance Vector Routing protocol, each router tries to find the shortest path to all other routers in the system in a distributed manner. The highlights are summarized below.

- **Routing Table**: Each router A maintains a routing table that includes one entry for every router in the system including itself. Each entry consists of the following fields: (i) the remote router id (say B); (ii) the cost of the path to router B; and (iii) the next hop to get to router B. Additional fields may be maintained depending on variations of the protocol.

- **Initialization (bootstrap):** The router is initialized with a set of neighbors and the cost of the links to the neighbors.

- **Routing Update Messages:** Each router periodically exchanges routing update messages with each of its neighbors.
  - When router A sends a routing update message to router B, it must for each router C (that exists in A's routing table) indicate the cost as well as the next hop that A uses to reach C. On receiving the update from A, B updates its table, for each router C that is being advertised by router A, in the following fashion:
  - If router B does not have an entry for C, then B must add an entry for C.
  - If it has an entry, then it uses the below two rules.
  - **Split Horizon Rule**: When B receives the update message from A, it checks whether the next hop that A advertises to reach C is itself (B). If so, the update pertaining to entry C is ignored. If not, B's route to C is updated if it is possible to obtain a better route by going through A.
  - **Forced update rule:** When B receives an update message from A, if B already uses A as next hop to reach C, and A now advertises a higher cost to C, then B must update its cost to reflect the higher cost.
  - (*Note*: You may assume that only periodic updates are to be supported. No triggered updates need to be implemented)

- **Router Failures:** Each router must check that it is receiving updates from its neighbors on a regular basis. If no updates have been received from a neighboring router for a certain amount of time, the link between them is marked as inactive. The normal distance vector operations will eventually lead to finding alternate paths to replace the invalidated routes.

Implementation:

In your implementation, each of the routers will run as a user-level process and each router binds to a distinct UDP port. The ***Network Emulator***, which will be given to you, binds to its own distinct UDP port as well. The Network Emulator maintains information regarding the entire network topology, provides each router with bootstrap information, and helps achieve communication between routers. The details regarding the interaction between the Network Emulator, and Router Processes are summarized below:

Network Emulator [you do not have to implement this]:

 - The Network emulator reads a configuration file that specifies the number of routers and links in the topology, information regarding which routers are connected by links, and costs of the links. The exact format of the configuration file is specified in a separate section.

Routers' Initialization:

 - When a router (to be implemented in file **router.c**) starts up, it sends an INIT_REQUEST message to the Network Emulator, which includes only its router-id. The router's ID should be between 0 and **MAX_ROUTERS** – *1*, where **MAX_ROUTERS** is the maximum number of routers that the system can support. This ID is fed by you in the command prompt as an argument when you run your router binary so **make sure you supply consistent values**.
 - The Network Emulator waits until receiving an INIT_REQUEST from **ALL** routers (to ensure all routers are alive before exchanging messages), after which it sends each router an INIT_RESPONSE message that includes information regarding the neighbors of the router, and the cost of the links to the neighbors. The Network Emulator stores a mapping between the router id and router port so it can properly forward any packet tagged with a destination router id.

Routers' Failures:

 - It is possible that a router fails (or the router process is explicitly killed). If no updates have been received by a router *A* from a neighbor *B* for **FAILURE_DETECTION** seconds, then, the router *A* marks the link to *B* as inactive. *A* must then modify its table to indicate that any route for which *B* is the next hop is no longer valid and has infinite cost.
 - The Network Emulator is implemented such that it responds with the appropriate INIT_RESPONSE whenever the dead (killed) router is restarted at a later point.
 - Upon the receipt of INIT_RESPONSE, a router initializes its routing table with costs to neighbor routers. The known entries of the routing table will grow as the router gains knowledge of other non-neighboring routers in the system.

Routers' Updates:

Each router periodically, every UPDATE_INTERVAL seconds, sends an RT_UPDATE message, including its complete routing table, to neighboring routers. To achieve this, each router sends its RT_UPDATE message to the Network Emulator. The Network Emulator in turn checks **only** the dest id of the packet and forwards the message to the appropriate destination router. The emulator itself does not duplicate any message; it simply ensures an incoming packet is sent to the correct neighbor of the sender.

Convergence:

We consider a routing table to have converged if it has not been modified for CONVERGE_TIMEOUT seconds, even though several update messages may have been received.

Packet Types and Formats:

The following packet types must be exchanged:

INIT_REQUEST	Sent by a router to the Network Emulator when it starts up. The router sends its id, and requests information regarding neighboring routers and link costs
INIT_RESPONSE	Sent by the Network Emulator to the router in response to the INIT_REQUEST, containing information regarding the router's neighbors and link costs
RT_UPDATE	Each router periodically sends a route update message to its neighbors. Note that these messages are sent to the Network Emulator, which in turn forwards to the appropriate router. A route update message sent by router A to router B includes an entry for every router in the network that A knows, and for each entry, the appropriate distance vector information such as the next hop and cost

For detailed specifications regarding the format of each of these messages, please refer to the header file **ne.h** that we are providing you with.

Binaries and Source Files:

To start this lab, go to blackboard and download the tar file lab2-files.

- **Files provided to you**:
  - router – The *Router* binary file based on our solution
  - ne – The *Network Emulator* binary file
  - router.h – Header file that defines functions to manipulate the routing table
  - ne.h – Header file that defines packet structures & functions to perform endian conversions
  - unit-test.c – Source file which contains unit test for the routing table update functions that you will implement
  - endian.c – Source file that has some useful routines for doing the endian conversions
  - Makefile – Make file that helps in compiling and building the *unit-test* and *router* binaries
- **Files you should implement:**
  - routingtable.c – Implement the functions to manipulate the routing table as defined in router.h
  - router.c – Implement the distance vector protocol using functions in routingtable.c

A complete example is also provided to help you debug your code. Your implementation **must be compatible** with our binaries, and with the header files that we have provided. **You are not allowed to make any changes to the header files**. The header files will guide your work, by providing more detailed information regarding useful functions and packet formats to be used.

Important Design Rules:

- **In `routingtable.c` you must implement** useful routines that **manipulate the routing table** of the router. The prototypes of these functions are defined in the file **router.h** that we are providing. **NOTE:** We won't accept your code if everything is implemented in router.c; use both routingtable.c and router.c. The functions that you implement in routingtable.c are stand-alone routines that can be tested for basic correctness using the unit-test code that we have provided. This means you have to build and run **unit-test** to test your logic to manipulate the routing table. The details of the functions are given below:
  - **InitRoutingTbl() :** This function initializes the routing table with the neighbor

3

information received from the *Network Emulator*. Adds a self-route (route to the same router) in the routing table.

- o **UpdateRoutes():** This function is invoked on receiving route update message from a neighbor. It installs in the routing table, new routes that are previously unknown. For known routes, it finds the shortest path and updates the routing table if necessary. It also implements the forced update and split horizon rules. It returns '1' if the routing table changes during the process and '0' otherwise.
- o **ConvertTabletoPkt():** This function is invoked on timer expiry to advertise the routes to neighbors. It fills the routing table information into a struct pkt_RT_UPDATE, which is passed as an input argument.
- o **PrintRoutes():** This function is invoked whenever the routing table changes. It prints the current routing table information to a log file that is passed as an input argument. The format of the log message to be printed is explained in detail in the section "Format of Router Log File".
- o **UninstallRoutesOnNbrDeath():** This function is invoked on detecting an inactive link to a neighbor (dead nbr). It checks all the routes that use the dead nbr as the next hop and changes the cost to INFINITY.

- ✚ Please note that you must define the following two global variables in **routingtable.c**. The variables must be used only in **routingtable.c**.
  - o **struct route_entry routingTable[MAX_ROUTERS]:** This is the routing table that contains all the route entries. It is initialized by InitRoutingTbl() and updated by UpdateRoutes().
  - o **int NumRoutes :** This variable tells the number of routes in the routing table. It is initialized by InitRoutingTbl() and updated by UpdateRoutes().
- ✚ Please **read through the comments in router.h carefully**, for further details on the functions and variables.
- ✚ **In router.c you must use select()** to monitor the UDP file descriptor and to implement the timer functionality. On receiving a RT_UPDATE packet, **UpdateRoutes** is called to modify the routing table according to the protocol. If the UPDATE_INTERVAL timer expires, first **ConvertTabletoPkt** is called and a RT_UPDATE is sent to all the neighbors. Second, we also check to see if any neighbor has not sent a RT_UPDATE for FAILURE_DETECTION seconds (3*UPDATE_INTERVAL). All such neighbors are marked as dead and **UninstallRoutesOnNbrDeath** is called to check for routes with the dead nbr as next hop and change the costs to INFINITY.
- ✚ **Your code must run on Linux**. Note that in Sun Solaris the select() has a different behavior than in Linux, so make sure you test your code on Linux. **Everything will be graded on Linux**.

Deliverables:

To successfully complete the project, you must implement and turn in only routingtable.c and router.c files. Your implementation must be compatible with our files. In addition, you need to adhere to the format of the configuration file described below, and have your routers produce logs in the manner we describe.

Configuration File Format:

The Network Emulator reads the topology information from a configuration file. The configuration file has the following format.

```
<Number of Routers>
<Router1> <Router2> <LinkCost>        ## Information about Link1
………                                   ## Information about other links, 1 line per link
```

For example, suppose we have the following topology
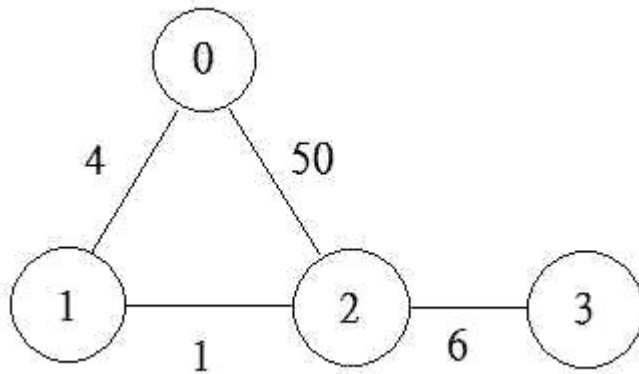
The configuration file would look like this:

4		# number of routers
0	1	4
0	2	50
1	2	1
2	3	6

Format of Router Log File:

Each router with id <id> **must** produce a log file **router<id>.log**. The routing table of the router must be printed to the log file **whenever the table changes** (e.g., when a new route is added to the table, or the cost, next hop, or path to an existing router changes). The log file contains the history of routing table changes for a given router. The routing table logged has to be in the format described below. If a routing table has not changed for **CONVERGE_TIMEOUT** seconds, append "x:Converged" to the end of the routing table, where x is the number of seconds (rounded down to the nearest integer), that has elapsed since the router receives an INIT_RESPONSE from the Network Emulator.

TIP: Use `fflush` *to ensure routing changes are immediately written to the log so even if a router is killed with Ctrl-C, the routing table changes are still logged.*
Note: Print your routing table only whenever there is a change. Otherwise we will deduct points if you keep printing it continuously every second or so.

The format is as follows:

Routing Table:
```
SRC -> DEST: NEXT_HOP, COST
...   # 1 line for each destination it knows
```
*Note: if there is an entry with INFINITY cost in the routing table, print out this entry with (i) the cost being the value of INFINITY defined in **ne.h**; and (ii) the next hop being any router.*

Examples:
router 0 would have the following routing table at bootstrap
```
Routing Table:
R0 -> R0: R0, 0
R0 -> R1: R1, 4
R0 -> R2: R2, 50
```

The converged table would look something like:

```
 Routing Table:
```

R0	-> R0: R0, 0
R0	-> R1: R1, 4
R0	-> R2: R1, 5
R0	-> R3: R1, 11
7:Converged	# The routing table hasn't changed for the last CONVERGE_TIMEOUT seconds.

Make sure the logs written to the file don't have unnecessary print statements. Please use the **DEBUG** flag that is provided to you in the *Makefile,* to add any extra debug statements in your code. By default this flag is turned off, so that only the required logs are printed.

 A set of log files for this 4-router topology is included in the example folder of the gz file for clarity.

How to run the router and ne:
- `router <router id> <ne hostname> <ne UDP port> <router UDP port>`

- `ne <ne UDP Port> <ConfigFile>`

Example of execution of 4 routers:

On the host **enad302pcX**	On the host **enad302pcY**

`./ne 9999  4_routers.conf`	`./router 0 enad302pcX 9999 9991`
	`./router 1 enad302pcX 9999 9992`
	`./router 2 enad302pcX 9999 9993`
	`./router 3 enad302pcX 9999 9994`

Unit Testing:

We have provided code that contains unit tests for the functions that you will implement in routingtable.c.

On running the unit-test code, you will get messages such as:

Test Case 1: PASS Initialize routing table

Test Case 1: FAILED to initialize routing table

The first message indicates that the test case passed and the second one indicates that it failed. You will get similar messages for each of the test cases. The string after the *PASS/FAILED* (e.g. *Initialize routing table*) will tell you what functionality is being tested. The command-line argument to run the unit test is `unit-test`

System Integration Testing:

In addition to testing your full implementation with our example, you should test your work as rigorously as possible. To help, we suggest a few possible test scenarios

- Small topology, no failures
- Large topology, no failures
- Failure of node: kill a router and see how routing tables change to reflect the new setting. Notice that killing certain routers might sometimes cause *count-to-infinity* problem. If *count-to-infinity* problem occurs, make sure the counting stops when the cost reaches the value of `INFINITY` defined in **ne.h**. Note that the relevant final routing table costs should have the the value of INFINITY defined in **ne.h**.
- Restart of node after failure: other routers update their routing tables to reflect possibly improved paths

Node failure is achieved by issuing an explicit kill command to the router process or Ctrl-C.

Makefile Hints:

- To compile and build the router binary, run

    make clean; make router
- To compile and build the unit-test binary, run

    make clean; make unit-test

Advice/Milestones:

- **This project requires dedication. Please work regularly right from the first day on this project and you will find it easy to do.**
- Do your work in a modular fashion. For example, check that your router can send and receive `INIT_REQUEST/INIT_RESPONSE` packets correctly, before implementing other functionality.
- Run ***unit-test*** and make sure your route update routines work fine before implementing the complete router framework.
- Here are some milestones for you for each week.

Week 1	Functions in routingtable.c & Contact *Network Emulator,* initialize routing table
Week 2	Unit test the functions in routingtable.c & Implement select() code
Week 3	Handle router failure scenarios & System Integration test
Week 4	Complete with rigorous testing of various failure cases.

- Note that the above weekly milestones are just suggestions to help you get in pace with the project. But we expect you to do beyond what is specified as a week's target.

Checkpoint Submission:

- For checkpoint we minimally require that you complete implementing the following
  - o Contact *Network Emulator* and initialize routing table with neighbor information
  - o Routing table update functionality in routingtable.c, which passes unit tests.
- Submit the following files with the above implementations.
  - o router.c
  - o   routingtable.c
- **Please note that the checkpoint submission will be graded and points will be included in the total grade for the project.**
- **For checkpoint, we will only grade routingtable.c. So you will NOT lose points if router.c is not complete or has errors.**
- **This is just a minimal requirement and we expect that by checkpoint every group would have done beyond what is specified, to be on track to complete the project.**

Final Submission:

A copy of the files **`routingtable.c`**, **`router.c`** and a text file **<login1>.<login2>.txt** must be submitted using the " *turnin*" command where login1 and login2 are login names of the two members of the group. Only one member per group needs to perform the submission. Multiple submissions are allowed (before the due-date). The text file should describe the status of your submission (i.e. whether all parts are working, if not, which parts are working, and where you think the problem is, in the parts that do not work).

Submission instructions:

Log into `ecegrid.ecn.purdue.edu`
Move to the same directory as the file you are going to
submit. Execute the following:

For checkpoint:
```
turnin -c ece463 -p lab2checkpoint routingtable.c router.c
<login1>.<login2>.txt
```
For final project:
```
turnin -c ece463 -p lab2 routingtable.c router.c
<login1>.<login2>.txt
```