ECE 463 - Introduction to Computer Networks Fall 2019

Project II: Distance Vector Routing

Checkpoint Due	**10/31/2019, Thursday, 9:59 PM**
Due Date	**11/14/2019, Thursday, 9:59 PM**

To be done individually or in groups of two

Policies for this project

These policies are in addition to any other policies in the course syllabus.

Collaboration Policy: This project is to be done individually or in groups of two. Only one submission is needed for each group.

ABET Policy: You must score 40% on the project overall to meet the ABET outcome and obtain a passing grade in the course.

Individual Assessments:

Both project members must contribute substantially to the project. We may conduct individual assessments (e.g., through individual interviews and code walk through) where we think necessary, or at random. If deemed appropriate, the grades assigned to each student in a group may be different to reflect individual contributions.

Disputes within a team:

We encourage team members to resolve any issues between themselves. If the issues cannot be resolved, please bring any concern with a team member to the attention of the instructor by the Project Checkpoint deadline to the extent possible. It will be much harder for the instructor to take any steps to help at a later stage.

Cut-off time for answering questions: To discourage last minute work, we will stop answering questions 2 hours before the deadline. (i.e., 8pm Tuesday on each of the days of the deadline). Please plan accordingly.

Late Policy

1. There will be a 12-hour grace period. That is, for the first 12 hours after the deadline, no penalty will be imposed. No questions will be answered during this grace period.
2. Beyond 12 hours, no submission will be accepted and a score of zero will be awarded for the relevant parts.

Note on automated grading: Please make sure to thoroughly test your code as we will be using an automated test script and cannot look at your code to see where the error is. Due diligence in testing to capture bugs is part of the requirements for the project. Last minute poorly tested submissions may not pass our tests and could get a low score. Further, you **MUST** test your code with valgrind and ensure there are no errors. We may deduct credit for errors with valgrind even if all test cases pass. Please note that many errors are not deterministic, and even though your code may appear to run properly on your system, it may have bugs that get triggered during our testing. We encourage team members to do a code review of each other's code to catch such non-deterministic errors.

Overview:

The goal of this project is to implement a simplified version of the path vector protocol.

Path 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. For this lab, you will be implementing the Path Vector variant. 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. In a distance vector routing protocol, 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. In a Path Vector implementation, however, it is necessary to also include the full path required to reach the destination. For example, say A wants to send a packet to B. If this packet must be forwarded from A to C, from C to D, and then from D to B, A's routing table must contain the full path A → C → D → B.
- **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, next hop, and full path that it 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.
    - **Path Vector Rule**: When B receives the update message from A, it checks if A advertises B's router id anywhere in the path vector for C. 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. This is a generalization of the split horizon rule.
    - **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 path 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 included in the full path 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 destination 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 path vector information such as the cost, next hop, and full path

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
    - routingtable.c – Skeleton file including function declarations for required functions. PrintRoutes function included
    - 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 path vector variant of the distance vector protocol using function 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 information received from the *Network Emulator*. This routing table initialization should include a self-route (route to the same router) in the routing table.
- **`UpdateRoutes():`** This function is invoked on receiving a route update message from a neighbor. For routes that were previously unknown, it installs the new routes into the routing table. For known routes, it finds the shortest path and updates the routing table if necessary. It also implements the forced update and path vector rules. It returns '1' if the routing table changes during the process and '0' otherwise.
- **`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.
- **`PrintRoutes():`** This function is provided to you and should be 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".
- **`UninstallRoutesOnNbrDeath():`** This function is invoked on detecting an inactive link to a neighbor (dead nbr). It changes all routes that use the detected dead neighbor in their path vector and changes their 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.`**
    - **`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().
    - **`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 threads** to monitor the UDP file descriptor and to implement all time based functionality. The expected functionality of each threads is described below in the "Thread Implementation" section. Additionally, you **must not use select() or fork()** in your code. If the select() OR fork() function is found anywhere within your code, **you will not receive ANY credit**.

- **Your code must run on Linux**. As was the case in the first project, all code will be graded on ECN servers. Thus, it is expected that you will use these servers to build and test your code prior to submitting it.

Thread Implementation:

- This project requires the use of threads. It is expected that your program takes the proper steps to initialize by sending the INIT_REQUEST packet and receiving and processing the INIT_RESPONSE packet. From here, it is required that you instantiate two threads. A UDP file descriptor polling thread and a timer thread.

- In your UDP file descriptor polling thread, you are to wait to receive an RT_UPDATE packet. Upon receiving such a packet, **UpdateRoutes** should be called to modify the routing table according to the path vector protocol. When receiving routes, it is important to make note of the time of the last received update packet from a node to check for dead neighbors. Similarly, it is important to track the time of the last routing table update to detect convergence.

- In your timer thread, it is expected that you constantly monitor all time-based constraints and implement 'timers' to control the frequency and convergence of events. You must ensure that once the UPDATE_INTERVAL timer expires, first **ConvertTabletoPkt** is called, followed by an RT_UPDATE packet being sent to all neighbors. Additionally, you must monitor the activity of your neighbors, and ensure that if you do not hear from some neighbor for FAILURE_DETECTION seconds or more, you mark the neighbor dead. This should be accomplished by calling the **UninstallRouteOnNbrDeath** function. Lastly, you need to check for the convergence of your routing table, by checking that no changes have been made for CONVERGE_TIMEOUT seconds, and appropriately marking the log file as such.

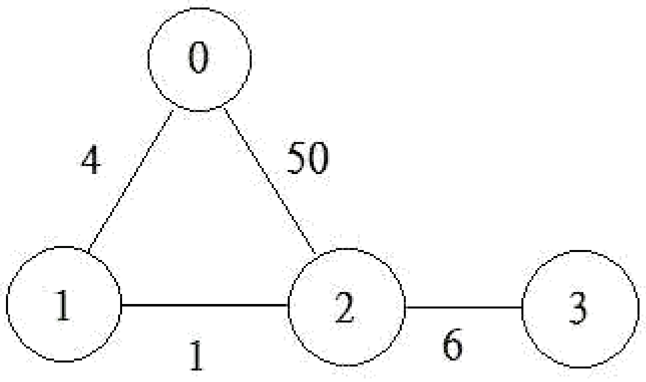
- Note that data will need to be shared between threads. Thus, it will be necessary to make use of mutex's to protect data from being overwritten.

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

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.

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 of the log is as follows and is implemented in PrintRotues for your conviencnce already. It is highly recommended that you do NOT alter this function:

```
Routing Table:
<<SRC> -> <DST>>, Path:<SRC> -> <NEXT HOP> -> … -> <DST>, Cost: <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 as defined in **ne.h**; (ii) the path vector printed is not important for this situation and does not need to be changed.*

Examples:
router 0 would have the following routing table at bootstrap

```
Routing Table:
<R0 -> R0>, Path: R0, Cost: 0
<R0 -> R1>, Path: R0 -> R1, Cost: 4
<R0 -> R2>, Path: R0 -> R2, Cost: 50
```

The converged table would look something like:

```
 Routing Table:
 <R0 -> R0>, Path: R0, Cost: 0
```

```
  <R0 -> R1>, Path R0 -> R1, Cost: 4
  <R0 -> R2>, Path: R0 -> R1 -> R2, Cost: 5
  <R0 -> R3>, Path: R0 -> R1 -> R3, Cost: 11
```

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 and network emulator on same host:

Network Emulator	Routers
./ne <2000+classID> 4_routers.conf	./router 0 localhost <2000+classID> <3000+classID>
	./router 1 localhost <2000+classID> <4000+classID>
	./router 2 localhost <2000+classID> <5000+classID>
	./router 3 localhost <2000+classID> <6000+classID>

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.
- 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	- complete functions in routingtable.c - contact the Network Emulator - initialize routing table
Week 2	- pass unit tests for routingtable.c functions - implement threading - verify convergence for basic routing cases
Week 3	- handle router failure scenarios - handle router restart scenarios
Week 4	- complete rigorous testing using various test 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
  - Contact *Network Emulator* and initialize routing table with neighbor information
  - Routing table update functionality in routingtable.c, which passes unit tests.
- Submit the following files with the above implementations.
  - routing.c
  - routingtable.c
- **Please note that points for the checkpoint submission will be included in the final grade of your project 2.**

- **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.**

Deliverables:

For each part, please follow the following rules to submit your work:

For part 1 (checkpoint):
- Zip the files “routingtable.c” and “router.c” using the following command:

  “zip lab2checkpoint.<login1>.<login2> routingtable.c router.c”
- Upload on blackboard in the folder “Project Submission” under “Lab 2 Part 1 (Checkpoint)”

For part 2 (final):
- Zip the files “routingtable.c” and “router.c” using the following command:

  “zip lab2final.<login1>.<login2> routingtable.c router.c”
- Upload on blackboard in the folder “Project Submission” under “Lab 2 Part 2 (Final)”

Please make sure the name follows the format above with BOTH of your login IDs listed.

Note: only one member per group needs to perform the submission.