UNIVERSITY COLLEGE LONDON
DEPARTMENT OF COMPUTER SCIENCE
3035/GZ01: Networked Systems
Individual Coursework 4: Implementing Distance-Vector Routing
Distributed: 12th December 2017; Due: 10th January 2018, 4:05 PM

In this coursework, you will write distance-vector routing code for a simple router. The coursework is worth a total of 100 marks, and represents 8 percent of your final grade for 3035/GZ01.

The most valuable reference for you to use while working on this coursework is the set of 3035/GZ01 lecture notes on distance-vector routing. Those slides contain a complete statement of the distance-vector routing algorithm in pseudocode, and examples of how the algorithm behaves on a variety of network topologies. You will also find it useful to refer to RFC2453 (RIP Version 2), which describes the Routing Information Protocol (RIP), a modern distance-vector routing protocol. Note that we expect you to implement the algorithm as presented in the lecture notes, including setting of the appropriate routing table entries' metrics to a reserved INFINITY value when a link goes down. We will also ask you to implement two further optimizations: split horizon with poison-reverse, and timeout-based expiration of routing table entries. Note that you are *not* to implement other "advanced" features of DV routing, such as triggered updates.

You must write your routing code in Java; the network simulator code we give you as a starting point for the coursework is written in Java.

All programming for this coursework must be done under Linux on the CS department's lab machines. We have ensured that the code we give you to use as a starting point works correctly on these lab machines. Note that these machines are accessible over the Internet, so you may work on the coursework either from home or in the labs.[1] The Linux lab machines are those with the following hostnames, all of which end in `cs.ucl.ac.uk`:

```
niagara frontal parietal temporal occipital sphenoid
ethmoid maxilla palatine zygomatic lacrimal
```

A Simple Network Simulator

A router isn't much good unless it's connected to other routers by links. If you were writing routing protocol software for a real, physical IP router, you could test the software by connecting several routers into a network topology, running your routing software on each of them, and observing whether users' data packets reach their destinations successfully.

Because it's not feasible for you to build a physical multi-router network to test your routing code, we'll do the next-best thing: we'll use "virtual" routers rather than physical ones. We provide you with code for a simple *network simulator* that models a set of routers connected together by a set of links. That is, the simulator reads a configuration file that describes a network topology, and it then simulates that network by running one copy of your routing code on each router, and passing packets between routers over the links listed in the configuration file.

[1] Because of the particular configuration of the CS department's network, if you would like to use any of the lab machines remotely, you must first log into a CS departmental gateway such as `newgate.cs.ucl.ac.uk` by `ssh`, then from there log into one of the lab machines using `ssh` again.

In this coursework, there is a rigidly defined interface for the router's code. In the simulator's configuration file, you specify not only the network topology, but the name of the compiled Java module for the router code you would like to run on each router in the network. In this way, you can actually mix different router implementations in a single simulated network!

We have provided you with a skeleton of the code for a distance-vector router, found in the file `DV.java`. You should implement your solution to the coursework by filling in the missing parts of this file. Do not change any of the constants that we've pre-defined in that file—they must be left as-is for the simulator to work properly. The skeleton adheres to the `RoutingAlgorithm` interface to the distance-vector router, which you *must not change;* any changes to the router's API will similarly cause the simulator not to work correctly! The skeleton code compiles, but all methods in it return dummy or null return values. To build your solution, you must implement the following in the file `DV.java`:

- the correct and complete bodies of the functions that are incomplete (return dummy or null return values) in the version of `DV.java` we've given you
- all methods in `DVRoutingTableEntry`, a class that implements the `RoutingTableEntry` interface found in `RoutingTableEntry.java`

Note that you should *not modify any files* in the coursework apart from `DV.java` and the configuration files for the simulator, described below. That is, all the code you write will go in `DV.java`.

The Java code we've given you is fully documented in Javadoc; to prepare the documentation, just type `make javadoc` in the directory containing your coursework files, and you will find the documentation for the code we've given you in a newly created `docs` subdirectory.

We have given you a `Makefile` (found in the set of files for the coursework) to help automate the compiling and running of your routing code and the simulator. A full description of the `make` utility is beyond the scope of this coursework. For the purposes of this coursework, all you need to know about compiling and running your code is:

- Don't modify the `Makefile`.
- To compile your routing code in `DV.java` into the compiled Java module `DV.class`, just type

  ```
  make
  ```

  in the same directory where the `Makefile` and all the source files are located.
- To run tests of your routing protocol implementation, after you've compiled your router's code with `make` as above, just type

  ```
  java Simulator config.cfg
  ```

  where `config.cfg` can be the name of any simulator configuration file (whose format we describe below).
- To see brief help on what functions the Makefile lets you automate, type:

  ```
  make help
  ```

Simulator Configuration File

Each time you run the simulator, it reads a configuration file that describes the particular network topology it should simulate, and any actions to take during the simulation (and when to take them), such as "take this link down after 15 seconds," "print out the routing table of this router after 32 seconds," *etc.* – as described further below.

Consider the following simple example configuration file:

```
updateInt 10

preverse off
expire off

router 0 2 DVsolution
router 1 2 DVsolution
router 2 2 DVsolution

link 0.0.1 1.0.1
link 1.1.1 2.0.1
link 2.1.1 0.1.1

send 10 0 1

downlink 10 1.1 2.0

uplink 12 1.1 2.0

dumpPacketStats 14 all

dumprt 14 all

stop 100
```

The first three non-empty lines of the above configuration file specify that all routers in the network should send DV protocol updates every 10 seconds, that split horizon with poison-reverse should be disabled at all routers, and that timeout-based expiration of routing table entries should also be disabled at all routers.

Thereafter, the configuration file describes a scenario involving three simulated routers with addresses 0, 1, and 2, arranged in a ring. 10 seconds into the simulation, router 0 originates a data packet (to be forwarded by the routers in the network) with destination address 1. Also 10 seconds into the simulation, the link between router 1 and router 2 goes down. This link comes back up 12 seconds into the simulation.

All routers dump summary statistics of how many packets they've sent, received, dropped, and forwarded after 14 seconds, and dump their routing tables after 14 seconds.

The simulation runs for 100 seconds.

Router IDs are simple integers, as are interface IDs on routers.

Now, let's fully define the syntax of lines in the configuration file.
The routing algorithm update interval is declared as follows:

`updateInt` *u*

where *u* is the interval between DV protocol updates for all routers defined thereafter in the configuration file. If **`updateInt`** is not declared in the configuration file, then all routers use a default update interval of 1 second.

One may also enable or disable specific DV protocol optimizations at all routers. Enable or disable split horizon with poison-reverse as follows:

`preverse` `on | off`

where *on* or *off* specifies whether or not the routing algorithm deployed on each router incorporates split horizon with poison reverse.

Similarly, one may enable or disable timeout-based expiration of table entries as follows:

`expire` `on | off`

where *on* or *off* specifies whether or not the routing algorithm deployed on each router should expire routing entries in its routing table based on a timeout interval.

A router is declared as follows:

`router` *id n classname*

where *id* is the integer ID of this router, *n* is the number of interfaces for the router, *classname* is the name of the compiled Java module that should be used for the routing software for this router.

A link is a connection between two routers. Links also have a metric in each direction (configured in real routers by the network administrator). Links are declared as follows:

`link` *r1id*.*r1if*.*r1w r2id*.*r2if*.*r2w* `[up | down]`

where *r1id* is the integer ID of the router at one link endpoint, *r1if* is the interface ID to which the link connects on *r1id*, *r1w* is the metric incurred by packets sent by *r1* on the link, and all the *r2** fields have the same meanings for the router at the other link endpoint. The last field in the **`link`** line is optional. If supplied, it defines the initial state of the link in the simulation as either `up` or `down`.

You control the length of the simulation with:

`stop` *time*

where *time* is the number of seconds to run the simulation (the clock starts at zero seconds).

To allow you to observe how packets are routed in the network, the simulator allows injecting data packets from a particular source to a particular destination. You can do so with:

`send` *time origin destination*

where *time* is the number of seconds into the simulation to originate the packet, *origin* is the ID of the router to send the packet, and *destination* is the destination ID to put into the packet.

To see how the routing system behaves when links go down and come back up, the simulator supports taking links down and up at specified times. To take a link down, use:

`downlink` *time router1.interface1 router2.interface2*

where *time* is the time to break the link, *router1* and *interface1* are the router ID and interface ID of one end of the link, and *router2* and *interface2* are the router ID and interface ID of the other end of the link. Note that the first router specified should be the one with the smaller router ID.

Similarly, you can bring a link up with:

`uplink` *time router1.interface1 router2.interface2*

where the parameters are the same as those for `downlink`.

Finally, the simulator supports two commands to let you inspect the internal state of routers at a specified point in time. To see how many packets have been sent (`s`), received (`r`), dropped (`d`), and forwarded (`f`) by a router, use:

`dumpPacketStats` *time router*`|all`

where *time* states when you'd like packet statistics, and either *router* specifies the ID of a single router where you'd like packet statistics, or `all` specifies that you'd like packet statistics from all routers.

To see a router's routing table, use:

`dumprt` *time router*`|all`

where the parameters are the same as those for `dumpPacketStats`. (Note that the function that outputs a router's current routing table is unimplemented in the initial code we've given you. You must implement this functionality; we define the format in which you must output routing tables in the next section.)

`dumprt` and `dumpPacketStats` will be very useful to you in debugging your router—if your router doesn't behave as you expect it to, you can add these commands to a simulator configuration file to view the routing tables and packet statistics at any step in time you like.

Completing the Coursework

The first step in getting started with the coursework is to make yourself a copy of the files we give you to start from. To do so, while in some directory under your home directory while logged into a CS lab machine, execute the following command:

```
tar vzxf ~ucacmha/gz01-2017/cw4.tar.gz
```

You will then find a new directory `gz01-cw4` in your current directory, which contains all the coursework files.

When we mark your coursework, we will use a series of tests for your router. In each test, we will run the simulator with your routing code on every router, using a configuration file with a test network topology.

There are several topologies on which we'll test your router. We've given you five of them: these are in the simulator configuration files named `test1.cfg` through `test5.cfg`. We describe these test cases in more detail further below. There are additional test cases with which we'll test your router, too, known only to the course staff. We hold these in reserve until marking time so that you have an incentive to make sure your router truly works correctly for all topologies, rather than trying to "target" your implementation to the five tests we've given you.

There are three stages in which you should complete this coursework. In each stage, there is functionality you must implement, and the functionality is cumulative across the three stages.

Stage 1: Baseline DV

In this stage, you must implement a baseline DV routing algorithm. No separate design document is required, but you must comment your code thoroughly, to fully explain how it works.

The first two tests, `test1.cfg` and `test2.cfg`, check the correctness of your baseline DV router implementation. They therefore disable split horizon with poison-reverse and timeout-based table entry expiration. *Leave these two features disabled in these two tests' configuration files! You will be marked based on your router's behavior on these two tests with these two features disabled.*

Stage 2: Add Split Horizon with Poison-Reverse

In this stage, you must add split horizon with poison-reverse (SH/PR) to improve the convergence behavior of your DV routing implementation. We've given you two test cases, `test3.cfg` and `test4.cfg`, that have `preverse` set to `on`. That's the configuration in which we will use these two tests when marking your submission, but the first part of this stage is to explore how your baseline router *without* SH/PR behaves on these topologies.

> **Written question (answer to be turned in):** Without SH/PR enabled, run `test3.cfg` and `test4.cfg`. Consider the link failures that occur during these simulations. When links fail, what routing pathologies, if any, do you observe in each of these two test configurations when DV does not use SH/PR?

Now implement SH/PR. Note that you will need to enable or disable SH/PR in your router in accordance with the setting of the **`preverse`** flag in the configuration file. The code we've given you reads that statement from the configuration file already; read the Javadoc documentation to see which variables to access in the DV implementation to obtain the value of this flag. (When testing your SH/PR implementation, be sure the **`preverse`** flag is `on`! And similarly, if you want to run without SH/PR after implementing them, be sure to turn this flag `off`.)

> **Written question (answer to be turned in):** Now with SH/PR enabled, examine the behavior of your DV implementation on `test3.cfg` and `test4.cfg`. For each of the two tests, does SH/PR prevent each pathology you previously observed? Explain why or why not.

(The submission instructions later in this document describe how to submit your answers to these questions with your code.)

Stage 3: Add Expiration of Stale Table Entries

In this stage, you must enhance your DV implementation further to expire routing table entries as necessary. Note that the DV algorithm described in lecture allows routing table entries to persist indefinitely. But if a destination goes down and stays down, routers do not need to maintain routing table entries for that destination. In fact, deleting such entries from routing tables will reduce the size of subsequently exchanged routing messages. You will need to enforce a deadline after which stale entries for unreachable destinations should be removed from the routing table. We require that you remove stale entries in accordance with the timeout policy specified in RFC2453 (which you will find a very useful reference). The RFC expresses

when an entry should be removed as a function of u, the update interval. Note that part of the mechanism specified in the RFC handles the loss of routing update packets. Your simulator, however, never drops routing update packets, so you do not need to incorporate mechanisms for dealing with such losses. Finally, note that expiration of entries for unreachable destinations helps reduce the size of the routing table and subsequent announcements, but doesn't hasten convergence.

`test5.cfg` tests for the removal of stale routing table entries. This test enables both SH/PR and stale table entry expiration—the configuration in which we will test your code on `test5.cfg` during marking.

Further Tips on Implementation and Testing

To facilitate automated testing of your router, you *must* adhere to the following format when you implement the `dumprt` command:

- Output, on one line:

  `Router` *n*

  where you replace *n* with the integer address of the router.

- For each destination in the router's routing table, print out one line, in the format:

  `d` *destid* `i` *intid* `m` *metric*

  where you replace *destid* with the integer address that is the destination for this entry, *intid* with the integer interface ID for this entry, and *metric* with the integer metric for this entry.

Your router should not output any text other than the above as part of its `dumprt` output. We use automated scripts to test your router's behavior, and these scripts rely on your strict adherence to the above output format.

You will be marked on whether you pass the five public router tests and course staff's private router tests, and on the clarity of your design and comments.

To help you check whether you've implemented your router correctly, we've *also* given you a correct and complete solution to the coursework that implements the requirements of all three above stages. Obviously, we cannot give you the solution in source-code form, nor in Java .class file form, which is fairly easily reverse-engineerable to source code. Thus, we've given you a specially prepared *compiled* version of the solution that you can run, but whose code you cannot see.

> **IMPORTANT:** For the model solution for the DV router to work correctly, you **must** execute the following command while in the directory containing all your coursework files:
>
> `LD_LIBRARY_PATH=`pwd`` if your shell is bash, or
> `setenv LD_LIBRARY_PATH `pwd`` if your shell is csh or tcsh

N.B. that in the above text, the single-quote marks are backquotes, not apostrophes. If you encounter error messages about shared libraries when you attempt to run the simulator, you are almost certainly incorrectly using apostrophes, when backquotes are what is called for. (The difference is in which way the mark slants; a backquote goes from upper-left to lower right.)

To prepare the model solution, type the command:

```
make gcj
```

Note that you only need do so once.
To run the model solution, do the following:

- Make sure you've set the `LD_LIBRARY_PATH` variable in your shell as explained above.
- Edit the configuration file you would like to try with the model solution. In the configuration file, on each `router` line, change the name of the Java module to run for that router to be `DVsolution` rather than `DV`.
- Type the command:

  ```
  ./Simulator config.cfg
  ```

  where `config.cfg` is the configuration file you would like to run.

To see how a correct DV implementation should behave (with and without SH/PR and/or expiring routing table entries, based on how you set these flags in the simulator configuration files), you can run the tests with the model solution. You can even try to run tests where some of the routers run the solution and some run your code, by setting routers' Java module name strings in the simulator configuration file accordingly. Note that you must run the simulator with `./Simulator` if you want to use the model solution on any routers; the model solution cannot be used on any router when you run the simulator with `java Simulator`.

Marking Scheme

Out of 100 marks in total for the coursework, we will allocate marks as follows:

- passing `test1.cfg`: 5 marks
- passing `test2.cfg`: 10 marks
- passing `test3.cfg`: 15 marks
- passing `test4.cfg`: 15 marks
- passing `test5.cfg`: 15 marks
- passing two unseen tests: 15 marks
- your written answers to the two questions (5 marks each): 10 marks
- coding style and comment clarity and completeness: 15 marks

Testing Your Code

You can use `make` to run the tests. Do so by typing:

```
make test1
make test2
make test3
make test4
make test5
```

to run each of the tests in the five test simulator configuration files, respectively. These `make` commands will store the output of the simulator in files named `testNOutput.txt`, where `N` is the number of the test in question.

What to Turn In, and How

Submission of CW4 is electronic using the CW4 page on the 3035/GZ01 Moodle web site. You must turn in both of the following files electronically:

- A **single** Java source code file, `DV.java`, containing your full implementations of *both* `class DV` and `class DVRoutingTableEntry`.
- A text file named `answer.txt` containing your answers to the two questions in Stage 2.

As submission is purely electronic, you do not need to turn in a hardcopy coursework cover sheet for this coursework.

Academic Honesty

You are permitted to discuss the concepts of distance-vector routing (that is, the lectures' and assigned readings' content) with your classmates, and to discuss debugging strategies with one another, but *you are not permitted to show your code to any other student, in whole or in part, nor are you permitted to contribute any lines of code to any other student's solution.* As one always does in an academic setting, you must acknowledge the work of others explicitly. In this case, that means that if a classmate discussed distance-vector routing with you, or helped you strategize on how to debug your code, you must state the identity of that classmate in what you hand in, and describe how they contributed to your work (clearly indicated in a comment at the top of your Java source code file `DV.java`).

All code that you submit must be written entirely by you alone.

We use sophisticated copying detection software that exhaustively compares code submitted by all students from this year's class and past years' classes, and produces color-coded copies of students' submissions, showing exactly which parts of pairs of submissions are highly similar. **Do not copy code from anyone, either in the current year, or from a past year of the class.** You **will** be caught, just as students have been caught in years past.

Copying of code from student to student is a serious infraction; it will result in automatic awarding of zero marks to all students involved, and is viewed by the UCL administration as cheating under the regulations concerning Examination Irregularities (normally resulting in exclusion from all further examinations at UCL). You have been warned!

Questions and Piazza Site

If you have questions about the coursework, please don't hesitate to visit us during office hours, or to ask questions on Piazza. When asking questions on Piazza, please be careful to mark your question as private if it reveals details of your solution. (Questions that don't reveal details of your solution, such as those about how to interpret the coursework text or lecture material, should be left public, though, so that all in the class may benefit from seeing the answers.)

As always, please monitor the 3035/GZ01 Piazza site. Any announcements (*e.g.,* helpful tips on how to work around unexpected problems encountered by others) will be posted there.