

Coursework 2: Distance-Vector Routing

Due date: 12 noon, 10th April, 2007

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 20 percent of your final grade for 6007/GC15.

The most valuable reference for you to use while working on this coursework is the set of lecture notes for the 6007/GC15 lecture on distance-vector routing. Those slides contain a complete statement of the distance-vector routing algorithm in pseudocode, and many examples of how the algorithm behaves on a variety of network topologies. 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. Note that you are *not* to implement "advanced" features of DV routing, such as poison-reverse, split horizon, or 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 build upon 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. The Linux lab machines are those with the following hostnames, all of which end in `cs.ucl.ac.uk`:

```
achebe akutagawa alcott alkali aquinas atwood austen bellow boell  
bronte calvini carey cervantes choukri cocteau collins cowper dahl  
dumas eco faulkner flaubert fontane gibbon gogol golding grass hardy  
heine hesse kahiga kerouac levi mahfouz nabokov okri proust pushkin  
sands seishonagon steinbeck swift tanizaki amritsar bhopal bhuj  
calcutta darjeeling delhi dwarka indore mumbai mysore patiala patna  
raipur madurai lucknow jhansi jaipur pune karwar
```

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 `aldgate.cs.ucl.ac.uk` by `ssh`, and from there, use the `rlogin` command to log into one of the above lab machines.

If you do not yet have an account on the CS lab machines, you are entitled to one as a student in this module. To get an account, visit the technical support office on the 4th floor of the CS department building, MPEB.

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 in accordance with the links listed in the configuration file.

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
- a routing table entry 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 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:

```
router 0 2 DVsolution 10
router 1 2 DVsolution 10
router 2 2 DVsolution 10

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 above configuration file describes of three routers, with addresses 0, 1, and 2, arranged in a ring. Routers 0, 1, and 2 send DV protocol updates every 10 seconds. 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. A router is declared as follows:

```
router id n classname u
```

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, and *u* is the update interval between updates generated by this router.

A link is a connection between two routers. Links also have a metric in each direction (configured in real routers by the system 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 observe what routes are used in the network, there must be a way of 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.

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:

```
dumpprt time router |all
```

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

Note that `dumpprt` and `dumpPacketStats` are 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 vzxvf ~ucacbnk/gc15-2007/cw2.tar.gz
```

You will then find a new directory `gc15-cw2` in your current directory, which contains all the coursework files.

To complete the coursework, you must implement a correct DV router. No separate design document is required, but you must comment your code thoroughly, to fully explain how it works.

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.

To facilitate automated testing of your router, you *must* adhere to the following format when you implement the `dumpprt` 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* is the integer metric for this entry.

Your router should not output any text other than the above as part of its `dump` output.

There are several topologies on which we'll test your router. We've given you three of them: these are in the simulator configuration files named `test1.cfg` through `test3.cfg`. The remaining topologies on which we'll test your router are 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 three tests we've given you.

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

To help you further in deciding if you've implemented your router correctly, we've *also* given you a complete and correctly functioning solution to the coursework. 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
```

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.

If you run the three tests with the solution code, you will be able to see exactly how a correct router implementation should behave. (You can even try to run tests where some of the routers run the solution and some run your code, by setting the router 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`.

Good luck!

Marking Scheme

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

- passing `test1.cfg`: 10 marks
- passing `test2.cfg`: 15 marks
- passing `test3.cfg`: 20 marks
- unseen tests: 30 marks
- coding style and comment clarity and completeness: 25 marks

Testing Your Code

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

```
make test1
make test2
make test3
```

to run each of the tests in the three 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

To receive full marks for this coursework, you must turn in all the following files:

- A **single** Java source code file, `DV.java`, containing your full implementations of *both* `class DV` and `class DVRoutingTableEntry`.
- Three test output files, `testOutput1.txt`, `testOutput2.txt`, and `testOutput3.txt`. These should be generated using the `make` commands described above.

After the above files are ready to hand in, you are to submit them electronically. To do so, you will use the standard CS department `handin` utility, as follows:

1. Log into a CS department lab machine.
2. Change to the directory where your completed coursework files are located.
3. Run the command `handin`.
4. When prompted for the course module code, enter `comp6007` or `gc15`, depending on which module you are enrolled in.
5. When prompted for the coursework code, enter `cw2`.

6. When prompted for the filenames to submit, enter *only* the exact four filenames listed above.
7. You will be given the opportunity to verify that the filenames you've entered are correct; if so, confirm your choice to complete submission.

If you submit the wrong files or omit any of them, you may run `handin` more than once; the course staff will receive one timestamped copy of your files for each time you run `handin`. Note that we will *always* mark the *last* submission you make with the `handin` program before the coursework deadline.

Note that *no hardcopy submission of this coursework is required*.

Lateness Policy

You will be marked down 10 percent of the total 100 marks for every 24-hour period or part thereof you submit this coursework late, including weekends. That is, if you submit one hour past the deadline or 23 hours past the deadline, the maximum marks you can receive will be 90. If you submit 28 hours after the deadline, the maximum marks you can receive will be 80, and so on.

These lateness deductions will be made *after* marking your coursework. So if you're one day late, *e.g.*, you'd need to submit a perfect coursework to receive 90 marks.

Academic Honesty

You are permitted to discuss your code with your classmates, and to help one another debug. 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 your code with you, 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.

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). The course staff use extremely accurate plagiarism detection software to compare code submitted by all students and identify instances of copying of code; this software sees through attempted obfuscations such as renaming of variables and reformatting, and compares the actual parse trees of the code. Rest assured that it is far more work to modify someone else's code to evade the plagiarism detector than to write code for the assignment yourself!

Questions and Course Mailing List

If you have questions about the coursework, please don't hesitate to ask them by email. Please direct your questions to Adam Greenhalgh, the demonstrator who developed

this coursework, at `a.greenhalgh@cs.ucl.ac.uk`, and cc these emails to Brad Karp, at `bkarp@cs.ucl.ac.uk`.

Please monitor the course mailing lists, `{6007,gc15}@cs.ucl.ac.uk`, during the period between now and the due date for the coursework. Any announcements (*e.g.*, helpful tips on how to work around unexpected problems encountered by others) will be sent to the lists.