ANC Assessed Exercise Report

Michael Kilian - 1003819k

March 5, 2014

Contents

1 System Overview

The core of the system is made up of 3 classes:

1. **NetworkNode:** a class representing a node in the network. Each node holds a routing table and a set of links (with cost) which associate it with its adjacent nodes in the network.

2. **Network:** holds a collection of networks and defines necessary methods to manipulate the network, including causing exchanges. It implements the distance vector routing. All additions/changes to the network must be done through an instance of the Network class: an object using a Network instance does not require knowledge of the underlying NetworkNodes. A Network takes a file as input from which it builds itself.

3. **DistanceRoutingSimulator:** the main class which implements the command line interface. This is effectively a wrapper over a Network instance.

The above are contained in the *anc* package. In addition, the *events* package defines two events which may be queued to happen after a given number of iterations. These are:

1. **FailureEvent:** defines a failure of a link in the network.

2. **CostChangeEvent:** defines a change in link cost.

Both of these classes inherit from an abstract *NetworkEvent* class. An event is defined to occur between two nodes, which must be specified in the events constructor.

Nodes may be named by any String, without space. E.g. 'n1', 'node1' and '198.0.0.1' would all be valid node names. The format of an input network file must conform to the following:

1. The first line contains the names of the nodes in the network, each separated by a space.
2. Subsequent lines define the links in the network as a space-separated triple (n_1,n_2,cost).
3. There is an empty line after the list of links, after which any text is allowed as comments. These are not read in by the program, and exist only to aid in making the file human-readable by providing a brief description of the network.

In order to control the count-to-infinity problem, the max cost of a route is defined to be 256. Note that in each routing table the outgoing link is defined by the 'next hop' that must be taken from the current node. There are no concrete link objects in the system.

Instead of true broadcasting, a node updates by 'pulling' the routing tables from other nodes. During an exchange each node updates a copy of its routing table. When a node n1 pulls a routing table from another node n2, it pulls a copy of the table at the start of the iteration. In this manner, n1 will not accidentally receive any of the changes made to n2 in the current iteration before it should. At the conclusion of an iteration, the table for each node is replaced by the copy on which updates were made. When discussing the example networks in the next section we will discuss exchanges in terms of each node pulling data from its neighbours.

The system does not guarantee that nodes are updated in any particular order. The examples in this report assume an order of updating which may not occur on a particular execution of the simulator.

2 Example Networks

2.1 Network 1 - Normal Convergence

The first example network is shown in Figure 1. This network can reach stability after 3 exchanges (2 exchanges which result in changes plus 1 to establish stability has been reached). The initial routing tables contain only the links between nodes.

Table 1 shows the routing tables for the nodes after the first iteration. Many of the optimal routes have been found by this point, but the nodes at the edge of the network still do not have complete tables.

Table 2 shows the routing tables after the second iteration (only the routing tables which have changed have been shown). At this point stability has been achieved. A third iteration will result in no changes to routing tables but will confirm stability.

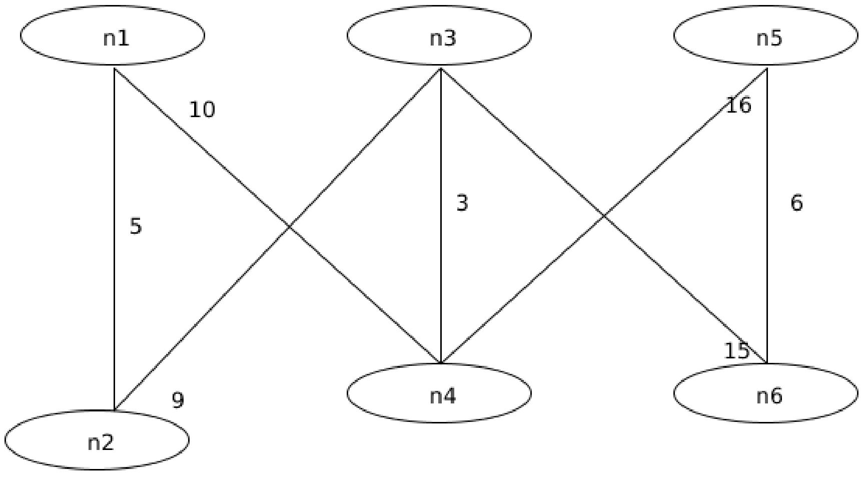

Figure 1: Example Network 1

2.2 Network 2 - Slow Convergence

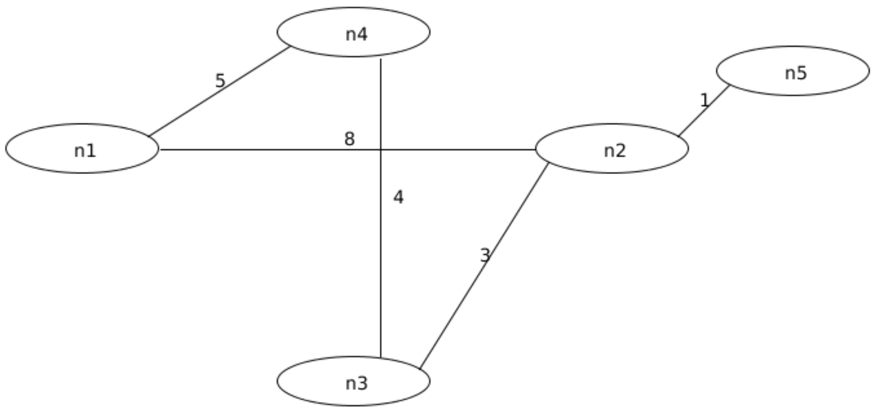

Figure 2: Example Network 2

Figure 2 shows the second example network. We will show the occurrence of slow convergence on this network with and without split horizon. We will cause the link from n2 to n5 to fail after iteration 4.

The network will stabilise after 3 iterations. We then purposefully cause another exchange to make the first link event fail then allow the exchange to run.

Node n1		
Destination	Distance (total cost)	Outgoing Link
n5	26	n4
n4	10	n4
n3	13	n4
n2	5	n2
n6	256	Unknown

Node n2		
Destination	Distance (total cost)	Outgoing Link
n1	5	n1
n5	28	n3
n4	12	n3
n3	9	n3
n6	24	n3

Node n3		
Destination	Distance (total cost)	Outgoing Link
n1	13	n4
n5	19	n4
n4	3	n4
n2	9	n2
n6	15	n6

Node n4		
Destination	Distance (total cost)	Outgoing Link
n1	10	n1
n5	16	n5
n3	3	n3
n2	12	n3
n6	18	n3

Node n5		
Destination	Distance (total cost)	Outgoing Link
n1	26	n4
n4	16	n4
n3	19	n4
n2	256	Unknown
n6	6	n6

Node n6		
Destination	Distance (total cost)	Outgoing Link
n1	28	n3
n5	6	n5
n4	18	n3
n3	15	n3
n2	24	n3

Table 1: Routing tables after first iteration.

2.2.1 Slow Convergence

Suppose the link from n2 to n5 has failed. We assume at this stage that n2 has immediately recognised its link to n5 is broken. The following sequence of

Node n1 (changed)		
Destination	Distance (total cost)	Outgoing Link
n5	26	n4
n4	10	n4
n3	13	n4
n2	5	n2
n6	28	n4

Node n5 (changed)		
Destination	Distance (total cost)	Outgoing Link
n1	26	n4
n4	16	n4
n3	19	n4
n2	28	n4
n6	6	n6

Table 2: Routing tables after second iteration.

events occur:

- n4 pulls the routing tables from its neighbours n1 and n2. The table from n2 shows that n5 is now unreachable through n2. However, n1 is advertising that n5 can be reached at cost 9 (n1-n2-n5). n4 concludes incorrectly that it can reach n5 through n1 at cost 13.

- n3 now pulls from its neighbours. In a similar manner, it recognises that n5 can no longer be reached through n2, but accepts the route advertised by n4. n3 now believes it can reach n5 through n4.

- Next n1 updates. It sees from n2's routing table that n5 can no longer be reached through n2. However, n4 is advertising a route through n1, which without split horizon, n1 will accept. n1 now falsely believes it can reach n5 through n4 at cost 15 (using the route n1-n4-n1-n2-n5, which clearly is not feasible.

- Finally, n2 updates. Although it knows that it can no longer reach n5 through its own link, its neighbours n1 and n3 are still broadcasting a route to n5. n2 will accept the route advertised by n3 (at a cost of 19, as opposed to the route advertised by n1 at a cost of 23).

A loop has now been created between n1 and n4. If the nodes continue to update in this pattern, the nodes in the route will continue to mistakenly believe that n5 can be reached. On each iteration the cost held by n1 will be increased, as it constantly must add a loop from itself to n4 and back to avoid the broken link from itself to n2 (step 3 is repeated). The network will only recover from this when the route to n5 advertised by n1 increases in cost to infinity (256 in this implementation). Once this occurs, n4 will recognise that n5 cannot be reached through n1 (the cost of the route advertised by n1 to n5 is infinity) and likewise advertise that n5 cannot be reached. This will eventually propogate to n3 and n2 over the subsequent iteration.

2.2.2 Split Horizon

In the above example, the slow convergence occurred because n1 was allowed to receive a route from n4 to n5 which involved the link from n1 to n4. In other words, n1 was fed a route which it had advertised itself, which was considered a better route than the infinite distance to n5 advertised by n2. With split horizon active, step 3 in the above example would not happen. Instead, n1 would not receive the route to n5 from n4 and continue to believe that n5 is unreachable. On the next iteration, n4 would pull this information from n1 and conclude there is no route to n5. n3 and n2 will then pick up on this respectively over the next two iterations.

3 Building and Running the Program

3.1 Requirements

The system is written in Java 6. As such a version of the Java Development Kit (JDK) and the Java Runtime Environment (JRE) are both required, such that the versions are compatable and support at least Java 6. For a Windows 7 machine, the easiest solution is to download and install the Java 7 compliant version, which can be found at *http://www.oracle.com/technetwork/java/javase/downloads/index.html* .

For the JRE & JDK respectively:

1. Click 'Download'
2. Accept the user agreement and download the correct version for the target machine (in this case, x64)
3. Run the resulting executable. From here an installation wizard will guide you through the remaining steps.

In addition, Apache ANT is required to build the executable. A tutorial specifying the full sequence required to install this on Windows 7 can be found at *http://madhukaudantha.blogspot.co.uk/2010/06/installing-ant-for-windows-7.html*.

3.2 Building the executable

To build the executable, simply do the following:

1. Extract the provided folder to the destination of your choice.
2. *cd* into the extracted folder.
3. Run *ant*.

This will produce an executable JAR file called *DistanceVectorRouting.jar*.

3.3 Running the program

Now you have built the required JAR file, you can run the program using:

```
java -jar DistanceVectorRouting <networkfile> [options]
```

The first argument is a file containing a network in the aforementioned format to use for the simulation. The only available option is '-s' which tell the simulator to use split horizon. This is, of course, optional. When ran successfully the program will immediately print a list of possible commands.

4 Status Report

Given more time there are a few areas in which the program could be improved.

The first would be the ability to determine the order in which the nodes update. As mentioned there is currently no guarantee regarding the order in which nodes are executed or ability to choose an ordering. This would be helpful if one wishes to simulate a very specific order of execution, such as that described in Section 2.2.1.

Secondly, there are a number of possible additions which would make it easier to see the current state of the network. This includes:

1. Printing out the current and past links on the network (i.e. see which links are active and which did exist but have been destroyed).

2. See the events which are queued up and after which iteration they are to occur.

Finally, it would be nice to add a GUI to the system which provided a visualisation of the network and showed the transmission update messages between nodes.