COMP 3331/9331: Computer Networks & Applications

Programming Assignment 2: Link State Routing

Due Date: 1 June 2008, 11:59 pm (Sunday) Assessment: 30 marks

Goal: In this assignment your task is to implement the link state routing protocol. Your program will be running at all nodes in the specified network. At each node the input to your program is a set of directly attached links and their costs. Each node will broadcast link-state packets to all other nodes in the network. Your routing program at each node should report the least-cost path and the associated cost to all other nodes in the network. Your program should be able to deal with dead nodes (i.e. nodes that fail).

Learning Objectives: On completing this assignment you will gain sufficient expertise in the following skills:

- Designing a routing protocol
- Link state (Dijkstra's) algorithm
- UDP sockets and network programming

Specification:

You will implement the following program:

Link State Routing (lsr)

- The program will accept the following command line arguments:
  - NODE_ID, the ID for this node. This argument must be a single uppercase alphabet (e.g., A, B, etc).
  - NODE_PORT, the port number on which this node will send and receive packets to and from its neighbours.
  - CONFIG.TXT, this file will contain the costs to the neighbouring nodes. It will also contain the port number being used by each neighbour for exchanging routing packets. An example of this file is provided below.

- Since we can't let you play with real network routers, the routing programs for all the nodes in the simulated network will run on a single desktop machine. However, each instance of the routing protocol (corresponding to each node in the network) will be listening on a different port number. If your routing software executes correctly on a single desktop machine, it should also work correctly on real network routers. Note that, the terms router and node are used interchangeably in the rest of this specification.

- Assume that the routing protocol is being instantiated for a node A, with two neighbours B and C. A simple example of how the routing program would be executed (assuming it is a Java program) follows:

  ```
  java lsr A 2000 config.txt
  ```

  where the config.txt would be as follows:

  2
  B 5 2001
  C 7 2002

The first line of this file indicates the number of neighbours. Following this there is one line dedicated to each neighbour. It starts with the neighbour id, followed by the cost to reach this neighbour and finally the port number. For example, the second line in the config.txt above indicates that the cost to neighbour B is 5 and this neighbour is using port number 2001 for receiving and transmitting link-state packets. The node ids will be uppercase alphabets and you can assume that there will be no more than 10 nodes in the test scenarios (however they need not be necessarily in alphabetical order), the link costs should be floating point numbers (up to the first decimal) and the port numbers should be integers. These three fields will be separated by a single white space between two successive fields in each line of the configuration file. The link costs will be static and will not change once initialised. Also, note that the nodes do not initially know the entire topology of the network; they only know the costs to their direct neighbours. You may assume that the configuration files used during testing will be in the appropriate format.

The rest of the specification has been broken down into 2 parts, beginning with the base specification as the first part and the subsequent part adding new functionality to the base specification. In order to receive full marks for this assignment you must implement both parts. Note that the bulk of the marks are allocated to the base specification. If you are unable to complete the second part, you will still receive marks for the first part. (The marking guidelines at the end of the specification indicate the distribution of marks).

Part 1: Base Specification

- In link-state routing, each node broadcasts link-state packets to all other nodes in the network, with each link-state packet containing the identities of the node's neighbours and the associated costs to reach them. You must implement a simple broadcasting mechanism in your program. Upon initialization, each node creates a link-state packet (containing the appropriate information – see description of link-state protocol in the textbook/lecture notes) and broadcasts this packet to its direct neighbours. The exact format of the link-state packets that you will use is left for you to decide. Upon receiving this link-state packet, each neighbouring router in turn broadcasts this packet to its own neighbours (excluding the router from which it received this link-state packet in the first place). This simple flooding mechanism will ensure that each link-state packet is propagated through the entire network.
- It is possible that some routers may start earlier than their neighbours. As a result, a situation may arise wherein the link-state packet is sent to a neighbour, which has not yet been instantiated. To overcome this, each router should wait for a random duration between 5 to 10 seconds before creating and broadcasting its own link-state packet. When we test your assignment, we will ensure that all nodes in the network are indeed up and running within 5 seconds of each other. You should make sure that you do the same when testing your programs. Also, note that the link costs are static and do not change.
- Each node should periodically broadcast the link-state packet to its neighbours every UPDATE_INTERVAL, which should be set to 1 second. In other words, the above process of broadcasting link-state packets throughout the entire network should repeat every second.
- You must use **UDP** as the transport protocol for exchanging link-state packets amongst the neighbours. Note that, each router can consult its configuration file to determine the port numbers used by its neighbours for exchanging link-state packets. Do not worry about the unreliable nature of UDP. Since, you are simulating multiple routers on a single machine, it is highly unlikely that link-state packets will be dropped. Further, the periodic broadcasting

of link-state packets will inherently make your protocol robust to occasional loss of link-state packets.

- On receiving link-state packets from all other nodes, a router can build up a topological view of the network. You may want to review your class notes and consult standard data structures textbooks for standard representations of undirected graphs, which would be an appropriate way to model this view of the network.

- Given a view of the entire network topology, a router can run Dijkstra's algorithm to compute least-cost paths to all other routers within the network. Each node should wait for a ROUTE_UPDATE_INTERVAL, which has a default value of 30 seconds, since start-up and then execute Dijkstra's algorithm. Given that there will be no more than 10 nodes within the network and a periodic link-state broadcast frequency of 1 second, 30 seconds is a sufficiently long duration for each node to obtain a complete view of the entire topology.
- Once a router finishes running Dijkstra's algorithm, it should print out to the terminal, the least-cost path to each destination node (excluding itself) along with the cost of this path. The following is the example output for node A in some arbitrary network:

```
least-cost path to node B: ACB and the cost is 10
least-cost path to node C: AC and the cost is 2.5
```

- We will wait for duration of ROUTE_UPDATE_INTERVAL after running your program for the output to appear (some extra time will be added as a buffer). If the output does not appear within this time, you will be heavily penalised. As indicated earlier, we will restrict the size of the network to 10 nodes in the test topologies. The default value of 30 seconds is sufficiently long for all the nodes to receive link-state packets from every other node and compute the least-cost paths.
- Your program should execute forever (as a loop). In other words, each node should keep broadcasting link-state packets every UPDATE_INTERVAL and Dijkstra's algorithm should be executed and the output printed out every ROUTE_UPDATE_INTERVAL. To kill an instance of the routing protocol, the user should type CTRL-C at the respective terminal.

Part 2: Dealing with Node Failures

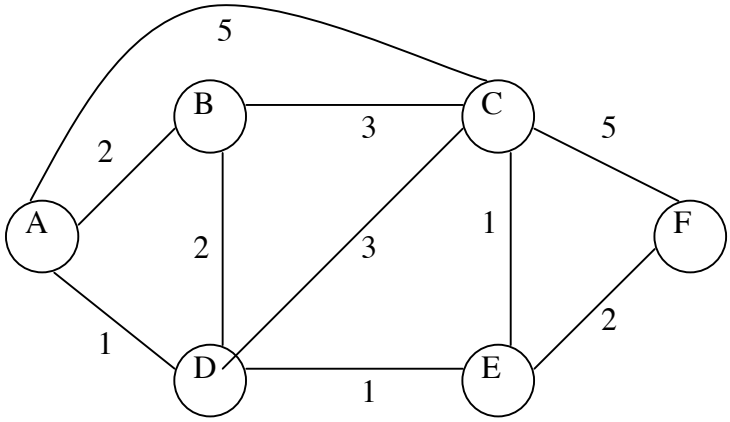
- In this part you must implement additional functionality in your code to deal with random node failures. Recall that in the base assignment specification it is assumed that once all nodes are up and running they will continue to be operational till the end when all nodes are terminated simultaneously. In this part you must ensure that your algorithm is robust to node failures. Once a node fails its neighbours must quickly be able to detect this and the corresponding links to this failed node must be removed. Further, the routing protocol should converge and the failed nodes should be excluded from the least-cost path computations.
- A simple method that is often used to detect node failures is the use of periodic *heartbeat* (also often known as *keep alive*) messages. A heartbeat message is a short control message, which is periodically sent by a node to its directly connected neighbours. If a node does not receive a certain number of consecutive hearbeat messages from one of its neighbours it can assume that this node has failed. Note that, each node transmits a link-state packet to its immediate neighbour every UPDATE_INTERVAL (1 second). Hence, this distance vector message could also double up as the hearbeat message. Alternately, you may wish to make use of an explicit heartbeat message (over UDP), which is transmitted more frequently (i.e.

with a period less than 1seconds) to expedite the detection of a failed node. It is recommended that you wait till at least 3 consequent hearbeat (or link-state) messages are not received from a neighbour before considering it to have failed. This will ensure that if at all a UDP packet is lost (UDP packet loss in a local network is very rare) then it does not hamper the operation of your protocol.

- Eventually, via the propagation of link-state packets, other nodes in the network will become aware that the failed node is unreachable and it will be excluded from the link-state computations (i.e. Dijkstra's algorithm).
- Once a node has failed, you may assume that it cannot be initialised again.
- While marking, we will only fail a few nodes, so that a reasonable topology is still maintained. Further, care will be taken to ensure that the network does not get partitioned. In a typical topology (recall that the largest topology used for testing will consist of 10 nodes), at most 3 nodes will fail. However, note that the nodes do not have to fail simultaneously.
- Recall that each node will execute Dijkstra's algorithm periodically after ROUTE_UPDATE_INTERVAL (30 seconds) to compute the least-cost path to every other destination. It may so happen that the updated link-state packets following a node failure may not have reached certain nodes in the network before this interval expires. As a result, these nodes will use the old topology information (prior to node failure) to compute the least-cost paths. Thus the output at these nodes will be incorrect. This is not really an error. It is just an artifact of the delay incurred in propagating the updated link-state information. To account for this, it is necessary to wait for at least two consecutive ROUTE_UPDATE_INTERVAL periods (i.e 1 minute) after the node failure is initiated. This will ensure that all the nodes are aware of the change in topology. While marking, we will wait for 2*ROUTE_UPDATE_INTERVAL following a node failure before checking the output.

An Example

- Lets look at an example with the network topology as shown in the figure below:



The numbers alongside the links indicate the link costs. The configuration files for the 6 nodes are available for download from the assignment webpage. In the configuration files we have assumed the following port assignments: A at 2000, B at 2001, C at 2002, D at 2003, E at 2004 and F at 2005. However note that some of these ports may be in use by another student logged on to the same CSE machine as you. In this case, change the port assignments in all the configuration files appropriately. The program output at node A should look like the following:

```
least-cost path to node B: AB and the cost is 2.0
least-cost path to node C: ADEC and the cost is 3.0
least-cost path to node D: AD and the cost is 1.0
least-cost path to node E: ADE and the cost is 2.0
least-cost path to node F: ADEF and the cost is 4.0
```

You may also test out the ability of your program to deal with node failures in the above example by causing node B to fail (for example).

Please ensure that before you submit, your program provides a similar output for the above topology. However, we will use different network topologies in our testing.

Sequence of Operation for Testing

The following shows the sequence of events that will be involved in the testing of your assignment. Please ensure that before you submit your code you thoroughly check that your code can execute these operations successfully.

1) First chose an arbitrary network topology (similar to the test topology above). Create the appropriate configuration files that need to be input to the nodes. Note again that the configuration files should only contain information about the neighbours and not of the entire topology. Work out the least-cost paths and corresponding costs from each node to all other destinations manually using Dijkstra's algorithm as described in the lecture notes (or textbook).

2) Log on to a CSE Linux machine. Open as many terminal windows as the number of nodes in your test topology. Almost simultaneously, execute the routing protocol for each node (one node in each terminal).

   ```
   java ls_routing A 2000 configA.txt (for JAVA)
   java ls_routing B 2001 configB.txt
   ```

   and so on. You may write a simple script to automate this process.

3) Wait till the nodes display the output at their respective terminals.

4) Compare the displayed paths and costs to the ones obtained in step 1 above. These should be consistent.

5) The next step involves testing the capability of your program to deal with failed nodes. For this choose a few nodes (max of 3 nodes) from the topology that is currently being tested (in the above tests) and terminate the nodes by typing CTRL-C in their respective terminal windows. Make sure that the nodes chosen for termination do not partition the network. Work out the least-cost paths from each node to all other destinations manually using Dijkstra's algorithm as described in the lecture notes (or textbook). Wait for a duration of 2*ROUTE_UPDATE_INTERVAL and observe the updated output at each node. Corroborate the results with the manual computations.

6) Terminate all nodes.

NOTE: We will ensure that your programs are tested multiple times to account for any possible UDP datagram losses (it is quite unlikely that your routing packets will be dropped).

Programming Notes

- Don't debug by embedding print statements in your code unless it is absolutely necessary. Instead, learn to use a symbolic debugger like *gdb* (or *jdb* for java). You can also find some nice GUI front-ends for *gdb* that make it easier to use. By using a debugger, you will save yourself tons of time finding the bugs in your code. Symbolic debuggers are used by professional programmers debug code so why not take advantage of this assignment to hone your professional skills. Note: Compile your code with the *-g* option in *gcc* so that the compiler will include debugging information in the executable. The javac compiler also has a *-g* option that does the same thing, but check the documentation.
- Since all nodes (i.e. routers) will actually be run on a single computer, you should use "localhost" as the destination IP address for the UDP packets exchanged between the nodes.
- Test your assignment out with several different topologies (besides the sample topology provided). Make sure that your program is robust to node failures by creating several failed nodes (however make sure that the topology is still connected). You can very easily work out the least-cost paths manually (as shown in the lecture notes or the textbook) to verify the output of your program.
- In writing your code, make sure to check for an error return from your system calls or method invocations, and display an appropriate message. In C this means checking and handling error return codes from your system calls. In Java, it means catching and handling *IOExceptions*.
- Do not worry about the reliability of UDP in your assignment. It is possible for packets to be dropped, for example, but the chances of problems occurring in a local area network are fairly small. If it does happen on the rare occasion, that is fine. Further, your routing protocol is inherently robust against occasional losses due to the fact that link-state packets are broadcast throughout the network. If your program appears to be losing or corrupting packets on a regular basis, then there is likely a fault in your program.

Additional Notes

- This is not a group assignment. You are expected to work on this **individually**.
- The program will be tested on CSE Linux machines. So please make sure that your entire application runs correctly on these machines. This is especially important if you plan to develop and test the program on your personal computers (which may possibly use a different OS or version).
- **Language and Platform**: You are free to use either C or JAVA to implement this assignment. Please choose a language that you are comfortable with. Your assignment will be tested on the **Linux** Platform. Make sure you develop your code under Linux (check the version on CSE machines).

Assignment Submission

We will inform you about the details of submission in 2 weeks (check the notice board on the website). You really only need one file: **lsr.c** (or **lsr.java**). If you are going to use any other files besides these two such as header files or other class files then you will have to submit a **Makefile** along with your code (only for C programs). This is because we need to know how to resolve the dependencies among all the files that you have provided. A link on how to create a *makefile* is available on the course website. Please ensure that the final executable is named *lsr*.

In addition you should submit a small report, **report.pdf** (no more than **3 pages**) describing the program design and a brief description of how your system works. Describe the data structure used to represent the network topology and the link-state packet format. Comment on how your program deals with node failures. Also discuss any design tradeoffs considered and made. Describe possible improvements and extensions to your program and indicate how you could realise them. If your program does not work under any particular circumstances please report this here. Also indicate any segments of code that you have borrowed from the Web or other books.

Late Submission Penalty: Late penalty will be applied as follows:

- 1 day after deadline: 10% reduction
- 2 days after deadline: 20% reduction
- 3 days after deadline: 30% reduction
- 4 days after deadline: 40% reduction
- 5 or more days late: NOT accepted

NOTE: The above penalty is applied to your final total. For example, if you submit your assignment 1 day late and your score on the assignment is 30, then your final mark will be 30 – 3 (10% penalty) = 27.

Plagiarism

You are to write all of the code for this assignment **yourself**. All source codes are subject to strict checks for plagiarism, via highly sophisticated plagiarism detection software. These checks may include comparison with available code from Internet sites and assignments from previous semesters. In addition, each submission will be checked against all other submissions of the current semester. Please note that we take this matter quite seriously. The LIC will decide on appropriate penalty for detected cases of plagiarism. The most likely penalty would be to reduce the assignment mark to **ZERO**. We are aware that a lot of learning takes place in student conversations, and don't wish to discourage those. However, it is important, for both those helping others and those being helped, not to provide/accept any programming language code in writing, as this is apt to be used exactly as is, and lead to plagiarism penalties for both the supplier and the copier of the codes. Write something on a piece of paper, by all means, but tear it up/take it away when the discussion is over. It is OK to borrow bits and pieces of code from sample socket code out on the Web and in books. You MUST however acknowledge the source of any borrowed code. This means providing a reference to a book or a URL when the code appears (as comments). Also indicate in your report the portions of your code that were borrowed. Explain any modifications you have made (if any) to the borrowed code.

Forum Use: You are free to discuss (and are in fact strongly encouraged to do so) issues relevant to the assignment on the course forum. However, refrain from posting large code-fragments on the forum. Students will be heavily penalised for doing so.

Marking Policy:

You should test your program rigorously and verify the results by trying out different topologies before submitting your code. Your code will be marked using the following criteria:

- Correct compilation of all files: **1 mark**
- Source code design (good structure and well commented): **3 marks**

- Correct computation of least-cost paths using Dijkstra's algorithm for several topologies: **16 marks**
- Appropriate handling of dead nodes, whereby the least-cost paths are updated to reflect the change in topology: **7 marks**
- Report: **3 marks**

IMPORTANT NOTE: For assignments that fail to execute **all** of the above tests, we will be unable to award you a substantial mark. Note that, we will test your code multiple times before concluding that there is a problem.