COMP 3331/9331: Computer Networks & Applications

Programming Assignment 2: Link State Routing

Due Date: 28 Oct 2016, 11:59 pm (Week 13) ***Marks: 10 + 1 bonus***

Updates to the assignment, including any corrections and clarifications, will be posted on the subject website. Please make sure that you check the subject website regularly for updates.

1. Change Log

Version 1.0 released on 19th September 2016.

2. Goal and Learning Objectives

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 nodes (i.e. neighbours) and the costs of these links. 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 failed nodes.

2.1 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 socket programming
- Handling routing dynamics

3. Assignment Specification

This section gives detailed specifications of the assignment. You can receive 1 bonus mark for submitting the assignment 1 week prior to the deadline.

3.1 Implementation Details

In this assignment, you will implement the link state routing protocol.

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 named Lsr.java) 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 (NOT the total number of nodes in the network). 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 that this neighbour is using for communication. 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, do not make assumptions that the node ids will necessarily start from the letter A or that they will always be in sequence. 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. Further, the link costs will be consistent in both directions, i.e., if the cost from A to B is 5, then the link from B to A will also have a cost of 5. You may assume that the configuration files used for marking will be consistent with the above description and devoid of any errors.

Important: It is worth restating that initially each node is only aware of the costs to its direct neighbours. The nodes do not have global knowledge (i.e. information about the entire network topology) at start-up.

The remainder of the specification is divided into two 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. 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 initialisation, each node creates a link-state packet (containing the appropriate information – see description of link-state protocol in the textbook/lecture notes) and sends this packet to all 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 nodes may start earlier than their neighbours. As a result, a node might send the link-state packet to a neighbour, which has not run yet. You should not worry about this since the routing program at each node will repeatedly send the link-state packet to its neighbours and a slow-starting neighbour will eventually get the information. That said, when we test your assignment, we would ensure that all nodes are initiated simultaneously (using a script).

Each node should periodically broadcast the link-state packet to its neighbours every UPDATE_INTERVAL. You should set this interval to 1 second. In other words,

Real routing protocols use UDP for exchanging control packets. Hence, 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. Furthermore, since link-state packets are broadcast periodically, occasional packet loss will not impact the operation of your protocol. <u>If you use TCP, a significant penalty will be assessed.</u>

On receiving link-state packets from all other nodes, a router can build up a global view of the network topology. 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 should 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 (the default value is 30 seconds) since start-up and then execute Dijkstra’s algorithm. Given that there will be no more than 10 nodes in the network and a periodic link-state broadcast frequency of 1 second, 30 seconds is a sufficiently long duration for each node to discover the global 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 an 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.

Restricting Link-state Broadcasts: Note that, a naïve broadcast strategy; wherein each node retransmits every link state packet that it receives will result in unnecessary broadcasts and thus increase the overhead. To elaborate this issue, consider the example topology discussed in the latter part of the spec. The link-state packet created by node A will be sent to its direct neighbours B, C and D. Each of these three nodes will in turn broadcast this link-state packet to their neighbours. Let us consider Node C, which broadcasts A's link state packet to B, D, E and F. Note that node B has already broadcast A's link state packet once (when it received it directly from A). Node B has now received this same link-state packet via node C. There should thus be no need for node B to broadcast this packet again. You MUST implement a mechanism to reduce such unnecessary broadcasts. This can be achieved in several ways. You are open to choose any method to achieve this. You must describe your method in the written report.

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. The other nodes should no longer compute least-cost paths to the failed nodes. Furthermore, the failed nodes should not be included in the least-cost paths to other nodes.

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 1 second) 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 then it does not hamper the operation of your protocol.

Once a node has detected that one of its neighbours has failed, it should update its link-state packet accordingly to reflect the change in the local topology. Eventually, via the propagation of the updated 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).

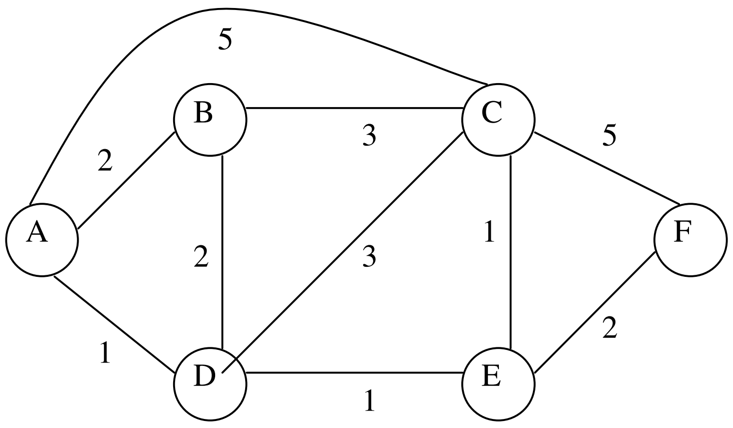
Once a node has failed, you may assume that it cannot be initialised again.

While marking, we will only fail a few nodes, so that a reasonable connected topology is still maintained. Furthermore, 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 an error. It is just an artefact 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 topology change. While marking, we will wait for 2*ROUTE_UPDATE_INTERVAL following a node failure before checking the output.

3.2. An Example

Let us 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
```

Note: It is not necessary that your program should print the paths to the destinations in alphabetical order.

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 (as an 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.

4. Additional Notes

This is not a group assignment. You are expected to work on this individually.

How to start: Sample UDP client and server programs are available on the Week 3 lecture material page. They are a good starting point to start your development. You will also find several links to network programming resources on that page.

Language and Platform: You are free to use one of C, JAVA or Python to implement this assignment. Please choose a language that you are comfortable with. The programs 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 programs on your personal computers (which may possibly use a different OS or version or JVM). Note that CSE machines support the following: **gcc version 4.9.2, Java 1.7, Python 2.7, 2.8 and 3. Note for Python:** In your report, please indicate which version of Python you have used. You may only use the basic socket programming APIs providing in your programming language of choice. Note that, the network will be simulated by running multiple instances of your program on the same machine with a different port number for each node. Make sure that your program will work appropriately under these conditions. See the sequence of operations listed below for details.

Error Condition: Note that all the arguments supplied to the programs will be in the appropriate format. The configuration files supplied as an argument to each node will also be consistent with the test topology. Your programs do not have to handle errors in format, etc.

You should be aware that port ID's, when bound to sockets, are system-wide values and thus other students may be using the port number you are trying to use. On Linux systems, you can run the command netstat to see which port numbers are currently assigned.

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 since the link state packets are exchanged every 1 second. If your program appears to be losing or corrupting packets on a regular basis, then there is likely a fault in your program.

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.

5. File Naming Convention and Assignment Submission

Your main program should be named **Lsr.c** (or **Lsr.java** or **Lsr.py**). You may of course have additional header files and/or helper files. If you are using C you MUST submit a makefile/script (not necessary with Java and Python). 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 and restricts excessive link-state broadcasts. 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.

You do not have to submit any topology files.

Here are the step-by-step instructions for submission:
1. Log in to your CSE account.
2. Create a directory called assign and copy ONLY the necessary files into that directory.
3. Tar this directory using the following command: “tar –cvf assign.tar assign”
4. Submit your assignment using the following command: “give cs3331 assign2 assign.tar”. You should receive a confirmation of your submission.

Alternately, you may submit the tar archive via the submission link at the top of the assignment web page.

Note that, the system will only accept *assign.tar* as the file name. All other names will be rejected. You can submit as many times as you like before the deadline. A later submission will override the previous submission, so make sure you submit the correct version. Do not wait till just before the deadline for submission, as there may be unforeseen problems (brief disconnection of Internet connectivity, power outage, computer crash, etc.).

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 2 days late and your score on the assignment is 10, then your final mark will be 10 – 2 (20% penalty) = 8.

6. 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. Do not post this assignment on forums where you can pay programmers to write code for you. We will be monitoring such forums. 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.

That said, we are aware that a lot of learning takes place in student conversations, and don't wish to discourage you from taking your classmates, provided you follow the Gilligan's Island Rule - After a joint discussion of an assignment or problem, each student should discard all written material and then go do something mind-numbing for half an hour. For example, go watch an episode of Gilligan's Island (or Reality TV in modern terms), and then recreate the solutions. The idea of this policy is to ensure that you fully understand the solutions or ideas that the group came up with.

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.

7. Forum Use

Students are strongly recommended to discuss about the assignment on the course forum. However, at no point should any code fragments be posted to the message forum. Such actions will be considered to be instances of plagiarism, thus incurring a significant penalty. Students are also encouraged to share example topologies that they have created to test their program.

8. 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). This will allow you to check that your program is computing the paths correctly.

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 lsr A 2000 configA.txt (for JAVA)
   java lsr 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 segment losses (it is quite unlikely that your routing packets will be dropped).

9. Marking Policy:

We will test your routing protocol for at least 2 different network topologies (which will be distinct from the example provided). Marks will be deducted if necessary, depending on the extent of the errors observed in the output at each node. After the marking process we will upload the test topologies on the website for all students to view.

Your code will be marked using the following criteria:

- Mechanism to restrict link-state broadcasts**: 1 marks**
- Correct operation of the link state protocol: **5.5 marks**
- Appropriate handling of dead nodes, whereby the least-cost paths are updated to reflect the change in topology: **2.5 marks**
- Report: **1 mark**

Bonus Mark: You may receive **1 bonus mark** for submitting the assignment a week before the deadline, i.e. by **21st October 2016 (Week 12)**. However, to receive the bonus mark, in addition to submitting by the early deadline, your code should have scored at least 7 marks (out of 10) as per the above criteria. The bonus mark can be used to offset lost marks in any assessable component in this course (e.g. mid-semester exam, final exam, labs, etc.).

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. You should test your program rigorously and verify the results by trying out different topologies before submitting your code.