The University of New South Wales

COMP3331/9331 Computer Networks and Applications

Assignment for Summer Session, 2019 (19T0)

Version 1.0

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

1. Change Log

Version 1.0 released on 11th January 2019.

2. Due date:

Due: 11:59pm Sunday, 3rd February 2019.

3. Goal and learning objectives

For this assignment, your task is to implement the link state routing protocol. Your program will be running at all routers in the specified network. At each router, the input to your program is a set of directly attached routers (i.e. neighbours) and the costs of these links. Each router will broadcast link-state packets to all other routers in the network. Your routing program at each router should report the least-cost path and the associated cost to all other routers in the network. Your program should be able to deal with failed routers.

3.1 Learning Objectives

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

1. Designing a routing protocol
2. Link state (Dijkstra's) algorithm
3. UDP socket programming
4. Handling routing dynamics

4. Assignment Specifications

This section gives detailed specifications of the assignment.

4.1 Implementation Details

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

Your program should be named **Lsr.py** (or **Lsr.java** or **Lsr.c**). It will accept the following 3 command line arguments:

- *ROUTER_ID*, the ID for this router. This argument must be a single uppercase alphabet (e.g., A, B, etc).
- *ROUTER_PORT*, the port number on which this router will send and receive packets to and from its neighbours.
- *CONFIG.TXT*, this file will contain the costs to the neighbouring routers. 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 routers in the simulated network will run on a single desktop machine. However, each instance of the routing protocol (corresponding to each router 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 router A, with two neighbours B and C. A simple example of how the routing program would be executed (assuming it is a Python program named Lsr.py) follows:

python Lsr.py A 5000 configA.txt

where A is the Router ID, 5000 is the port No for this Router A and configA.txt is the configuration file for Router A that has the following details:

```
2
B 6.5 5001
C 2.2 5002
```

The first line of this file indicates the number of neighbours for Router A. Note that it is not the total number of routers 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 configA.txt above indicates that the cost to neighbour B is 6.5 and this neighbour is using port number 5001 for receiving and transmitting link-state packets. The router 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 router 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 6.5, then the link from B to A will also have a cost of 6.5. 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 re-stating that initially each router is only aware of the costs to its direct neighbours. The routers 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. CSE students are to attempt both parts of the assignment while Non-CSE students are required to only attempt the base specifications. The marking guidelines are thus different for CSE and non-CSE students. These appear at the end of the specification indicating 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 router creates a link-state packet (containing the appropriate information – see description of link-state protocol in the textbook) 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 router should periodically broadcast the link-state packet to its neighbours every UPDATE_INTERVAL. You should set this interval to 1 second. In other words, a router should broadcast a link state packet every second.

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. **If you use TCP, a significant penalty will be assessed**.

On receiving link-state packets from all other nodes, a router can build up a global view of the network topology. 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:

I am Router A
Least cost path to router B: ACB and the cost is 14.2
Least cost path to router 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. You should be able to kill an instance of the routing protocol (e.g., 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 (Figure 1). The link-state packet created by node B will be sent to its direct neighbours A, C, D and E. Each of these three nodes will in turn broadcast this link-state packet to their neighbours. Let us consider Node C, which broadcasts B's link state packet to D. Note that node D has already broadcast B's link state packet once (when it received it directly from B). Node D has now received this same link-state packet via node C. There should thus be no need for node D 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 (For CSE students)

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).

You need to consider the case when a failed router joins back the topology again and starts sending link state update messages. Note that we are not considering the case when a previously unknown node joins the topology.

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.

4.1 Test Topology

You have been provided with configuration files for a sample topology of 6 routers. Use this topology to incrementally test your implementation. The correct output for Router A, before and after failure of Router D is given.

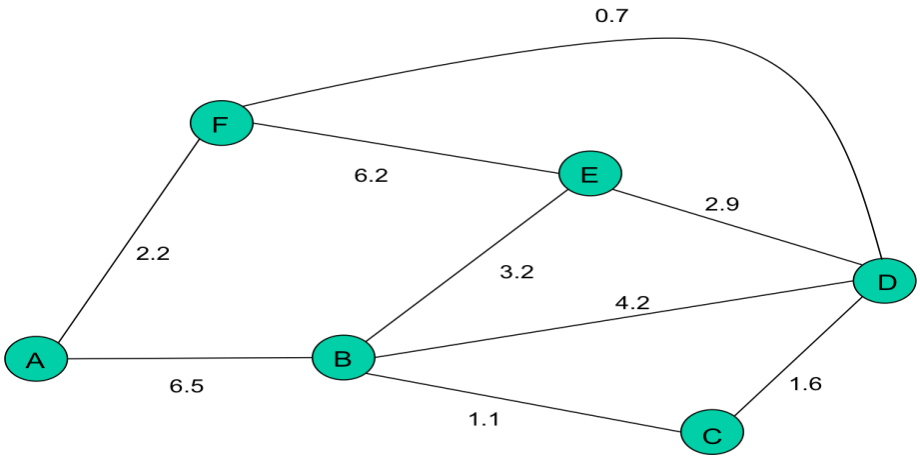

Figure 1: Test topology

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 these configuration files, we have assumed the following port assignments: A at 5000, B at 5001, C at 5002, D at 5003, E at 5004 and F at 5005. However, note that while testing your implementation on a CSE server, 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.

Following are the output for Router A before and after Router D has failed.

Output with all routers working:

```
I am Router A
Least cost path to router C:AFDC and the cost: 4.5
Least cost path to router B:AFDCB and the cost: 5.6
Least cost path to router E:AFDE and the cost: 5.8
Least cost path to router D:AFD and the cost: 2.9
Least cost path to router F:AF and the cost: 2.2
```

Output after Router D fails:

```
I am Router A
Least cost path to router C:ABC and the cost: 7.6
Least cost path to router B:AB and the cost: 6.5
Least cost path to router E:AFE and the cost: 8.4
Least cost path to router F:AF and the cost: 2.2
```

You are encouraged to post new topologies on WebCMS course discussion forum (config files) along with output for different routers so that other students can benefit from testing with different topologies.

Here is a useful link that can generate correct answers for various topologies.

https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html

5. Additional Notes

- This is NOT a group assignment. You are expected to work on this individually.
- **Language and Platform**: You are free to use 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 (i.e. your lab computers). We are unable to mark your assignment if it does not compile or run correctly on CSE lab computers, resulting in loss of significant marks**. 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 OS version or IDE). Note that CSE machines support the following: **gcc version 4.9.2, Java 1.7, Python 2.7, 2.8 and 3. If you are using Python, please clearly mention in your report**

which version of Python we should use to test your code. You may only use the basic socket programming APIs provided in your programming language of choice. You may not use any special ready-to-use libraries or APIs that implement certain functions of the specifications for you.

- 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 test 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 are free to design your own format and data structure for the messages. Just make sure your program handles these messages appropriately.
- Consider using multi-threading in your implementation. We recommend that you use at least three separate threads: for listening (for receiving LSA's), sending (for sending LSA's after every 1 sec) and Dijkstra's calculations (after every 30 sec). You may choose to use more than three threads as per your requirement.
- You are encouraged to use the course discussion forum to ask questions and to discuss different approaches to solve the problem. However, you should **not** post any code fragments on the forum.

6. Assignment Submission

Please ensure that you use the mandated file name. Your main program should be named **Lsr.py** (or **Lsr.java or Lsr.c**). You may of course have additional header files and/or helper files. If you are using C, then you MUST submit a Makefile/script along with your code (not necessary with Java or Python). This is because we need to know how to resolve the dependencies among all the files that you have provided. Beside the source code file, you should submit a small report, **report.pdf** (no more than 3 pages) see details in Section 7.

You can submit your assignment using the give command in an xterm from any CSE machine. Make sure you are in the same directory as your code and report, and then do the following:

1. Type tar -cvf assign.tar filenames e.g. tar -cvf assign.tar *.py report.pdf

2. When you are ready to submit, at the bash prompt type 3331

3. Next, type: give cs3331 assign assign.tar (You should receive a message stating the result of your submission). Use cs3331 even if you are enrolled in 9331.

Important notes

- The system will only accept **assign.tar** submission name. All other names will be rejected.
- **Ensure that your program/s are tested in CSE Linux machine before submission. In the past, there were cases where tutors were unable to compile and run students' programs while marking. To avoid any disruption, please ensure that you test your program in CSE Linux-based machine before submitting the assignment. Note that, we will be unable to award any significant marks if the submitted code does not run during marking.**
- You can submit as many times before the deadline. A later submission will override the earlier submission, so make sure you submit the correct file. Do not leave until the last moment to submit, as there may be technical or communications error and you will not have time to rectify it.
- Late submission penalty will be applied as follows:

    - 1 day after deadline: 20% reduction
    - 2 days after deadline: 40% reduction
    - 3 days after deadline: 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 10, then your final mark will be 10 – 1.5 (15% penalty) = 8.5.

7. Report

You have to submit a small report, **report.pdf** (no more than 3 pages) that must contain the following:

1. A brief discussion of how you have implemented the LSR protocol. Provide a list of features that you have successfully implemented. In case you have not been able to get certain features of LSR working, you should also mention that in your report.
2. 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.
3. Discuss any design trade-offs considered and made. List what you consider is special about your implementation. Describe possible improvements and extensions to your program and indicate how you could realise them.
4. Indicate any segments of code that you have borrowed from the Web or other books.

8. 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**. 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 (not complete modules/functions) 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 where 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.

9. 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 (either manually or use the provided link) using Dijkstra's algorithm. 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).

python Lsr.py A 5000 configA.txt (for Python)

python Lsr.py B 5001 configB.txt

and so on.

It is recommended that you 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 (or use the supplied link) using Dijkstra's algorithm. 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) Restart the failed node(s) and observe the least-cost paths after the network has converged.

7) 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).

10. Marking Policy

You should test your program rigorously before submitting your code. Your code will be marked using the following criteria:

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 router.

Your code will be marked using the following criteria:

Correct operation of the link state protocol (Base case only):

- CSE students: 7.5 marks
- Non-CSE students: 15 Marks

Mechanism to restrict link-state broadcasts:

- CSE students: 2 marks
- Non-CSE students: 2 marks

Appropriate handling of dead nodes, whereby the least-cost paths are updated to reflect the change in topology:

- CSE students: 5 marks
- Non-CSE students: Not marked

Correct handling of the case when a dead node joins back the topology:

- CSE students: 2.5 marks
- Non-CSE students: Not marked

Report:

- CSE students: 3 marks
- Non-CSE students: 3 marks