COMP9331

Assignment

By Rishabh Khurana

ZID-z5220427

Section-1

The python code implemented mainly mimics the following features of the LSR protocol: -

1) Least Cost calculation using Dijkstra's algorithm

2) Node failure detection

3) Controlled flooding of LSR packets

4) Update of Least cost path calculation upon node failure

Note: Update of Least cost path calculation upon node reboot of node/router has not been implemented in this version.

Note: executing ./start script on the terminal runs the default topology with Lsr.py file.

Note: The script uses python version 3 and above (python3)

The program has 3 main functions, namely 'send_heart_beat()', 'receive_messages()' and 'calc_least_cost()' along with some other functions which support the aforementioned main functions. All the main functions run in parallel using the threading functionality offered by python and an infinite while loop runs in each of the main functions.

The 'send_heart_beat()' function as the name suggests mainly sends an LSR packet with a delay of 1 second and updates the LSR packet upon node failure.

The 'calc_least_cost()' function is an implementation of Dijkstra's algorithm which runs every 30 seconds and calculates the least distance and the path from each node to the host router using the updated topology. It maps the current state of the Dijkstra's calculation into a dictionary called 'current_table' and updates it every time the least cost to a node has been calculated. A loop runs until all nodes from the 'current_table' have been removed and added to another dictionary known as 'shortes_dist' with the distance and path values.

The 'receive_messages()' function receives the message/LSR packet, updates the topology based on received message , and retransmits the message to its neighbours. It checks that the message is not sent back to a neigbour from which it was generated (heartbeat), and checks the packet recived from a router which has been broadcasted earlier should not be broadcasted again with the help of sequnce numbers. It also has a timer implemented in the function which runs for three seconds and checks the value of sequence number for each neighbouring node before the timer was initiated and after it passes 3 seconds. If any value of sequence numbers is same for neigbouring routers as before

then that neighbour is labelled as failed node, or else if all values remains same the timer is restarted. The failed neigbouring nodes are registered in the 'failed_nodes' list and the topology map is updated accordingly.

Section-2

The data structure used to represent the network topology is a dictionary of dictionaries. The network topology is stored in a variable 'topology' and an example of the format for the data structure is as follows:

{'F': {'A': 2.2, 'D': 0.7, 'E': 6.2}, 'A': {'B': '6.5', 'F': '2.2'}, 'E': {'B': '3.2', 'D': '2.9', 'F': '6.2'}, 'D': {'F': '0.7', 'B': '4.2', 'C': '1.6', 'E': '2.9'}, 'C': {'B': '1.1', 'D': '1.6'}, 'B': {'A': '6.5', 'C': '1.1', 'D': '4.2', 'E': '3.2'}}

In the above representation, the value of the distance between two nodes, for eg. A and F, can be accessed by simply plugging in the value of two nodes 'topology[A][F]' which will give us the distance 2.2 in this case.

The format for an LSR packet is as follows

Sequence_NO Router_ID Port_NO Neighbour1_ID distance1 Neighbour2_ID distance2.....

Firstly, we have the sequence number of the packet, which is initialised as 0 and gradually incremented for every second for a fresh packet that is to be sent (heartbeat message). The second variable is the router ID of the router from which the packet was generated, followed by the router ID of the neighbour and the distance of the router from its neighbour.

To deal with **node failures** I have implemented a 3 second timer in 'recv_messages()' function which compares the sequence number values of each neighbouring router before the timer was initialised and after 3 seconds had passed on the timer. If the sequence number values remain the same for any of the routers after 3 seconds have elapsed, then that neighbouring router is classified as a failed node. Once the router has detected a failed neighbour, it modifies it LSR packet and removes the data of the failed neighbouring node from that packet. This way gradually all routers get to know about the failed node and update their respective topology map which is stored in the variable 'topology'.

All the router IDs and the latest sequence number of LSR packet received from the respective router Id is mapped into a dictionary 'message_broadcasted'.

To **restrict excessive link-state broadcasts**, there is a mechanism implemented in 'recv_messages()' function which prevents any heart beat messages to be sent back to the neighbouring node from which it was generated.

Another mechanism which restricts excessive link-state broadcasts is also implemented in recv_messages() function , and makes use of sequence numbers. The router checks latest sequence number in the 'message_broadcasted' dictionary and if it receives the same packet from a different router with the same sequence number, generated from the same router ID, it supresses it's broadcast to its neighbours.

Section-3 & 4

None of the code has been borrowed from any other books or the web, however, the implementation of the Dijkstra's algorithm that is implemented in my code is inspired from the visual representation of Dijkstra's calculation for each step which is what makes the code **unique**. It captures the visual representation of each, and every iteration shown in the images in the lecture slides and maintains the current state of the Dijkstra's calculation in a dictionary called 'current_table'. However, I believe that there is a more efficient way to implement Dijkstra's algorithm with a much lesser time complexity using a priority queue. The priority queue would hold the distances of each router along with router ID and the router with minimum distance would be at the top of the queue. The priority queue implementation can be considered as **improvement or extension** to my existing code.