COMP3331 – Assignment Report

Farhan Sattar Ghazi (z5199861)

This assignment involved implementing the well-known routing protocol known as 'Link State Routing'. It is a protocol that is used for packet switching by routers.

The simple idea behind the routing protocol is as follows:

Every node (i.e router) in the network has an internal "link-state database" that it maintains for itself. Using this LS database, the router constructs a map of the entire network topology in the form of a graph which the router then uses to compute the shortest paths to every possible destination in the network by employing the well-known 'Dijkstra' algorithm.

My implementation

I have implemented the link state routing protocol employing the following features that I thought were helpful in making the protocol robust and receptive:

Multithreaded – I have 3 separate threads – Sender , Receiver and Heartbeat. Each thread helps accomplish a separate task. As their name suggests, the SendThread is used to broadcast the router LSA, the ReceiveThread receives LSA and processes them while HeartBeat is primarily used in node failure detection.

TimeStamps – In order to successfully determine dead nodes, I made use of the 'datetime' and 'timedelta' modules from the standard python library to timestamp each heart beat message. This proved effective as it enabled routers to easily detect with confidence which (if any) of its neighbours became inactive.

Router-specific Database – Each router has a local "link-state database" in the form of a sophisticated dictionary which it uses to construct the topology graph. This graph is then used to compute the shortest paths to every possible destination.

Unique Link-State Advertisement – My representation of the link state advertisement for each router is neatly packaged as a nested python dictionary which has a special 'FLAG' field that helps identify a packet that has been transmitted following the death of a node. This proved useful as it sped up the rate at which all my nodes converge.

UDP – The protocol uses UDP at the transport layer which ensures that the LSA's are transmitted at optimal speed.

My implementation covers both the base specification as well as the additional handling of random node failures, that is, on initialisation of routers, the routing protocol successfully creates the routing table for each router shortly after start up. In the event of a node failure, it also responds accordingly.

Data Structure Representation

I have represented the *network topology* as a nested list. Visually, it looks like this:

[[w, x , cost] , [y , z , cost] , ... , ...]

where w , x , y , z represent different nodes and the cost between them. This proved useful for both addition and removal of nodes as well as for the Dijkstra Computation.

As for the *link-state advertisement*, I represented them as nested python dictionary. Visually, it looks like this:

{ 'RID' : xxx , 'Port' : xxx , 'Neighbours Data' [{} , {} , {}] }

where, → the **RID** identifies who is the advertiser of this LSA
→ the **Port** idenfities the port number of this router
→ the **Neighbours Data** is a *python list of dictionary elements* that contain information about this routers immediate neighbours.

Restricting excessive link-state broadcasts

To reduce unnecessary broadcasts, I employed the following strategy:

Using pythons *set()*, I actively looked out for new LSA messages that have not been seen before by the router and add them to my set. This ensrues that my set would eventually contain all the nodes in the network (via flooding). Thereafter, if a router ever comes across a LSA that it has seen before (i.e belongs in the set), it would silently drop it (as the LSA had already been forwarded).

This ensures that routers don't unnecessarily keep broadcasting / forwarding link state packets.

Detecting and handling node failures

In order to detect node failures, I used python's datetime module to timestamp each heart beat message. Each router keeps track of the latest heart beat message received from each router in the network. Periodically, a function is triggered that computes

the difference between the last known record of a heart beat against the current heart beat received. If this difference grows large enough for a particular router, a router can then be marked as dead (i.e node has failed).

Once a router is declared to be dead, all neighbouring routers (who each have separate records of timestamps) transmit a fresh LSA to their active neighbours notifying them of the node that has failed. This new LSA in turn gets flooded across the network and eventually each router updates their state of the link-state database to account for the failed router.

Furthermore, the global graph object is updated and all redundant links (i.e to and from the dead router) are removed. As a result, in the next round of Dijkstra calculations, the dead node is omitted and the routing table reflects this change.

NOTE: This change in topology does take some time to propagate throughout the network but eventually all routers become up to date.

Design Trade-Offs

My implementation had a fair few trade-offs. The first being that, I decided to handle both the receiving of LSA's and consequent updates to my ADT's in the same thread (ReceiveThread). Ideally, I would have liked to have a separate thread for updating my data structures. However, in order to reduce the added complexity that comes with having multiple threads (i.e locks etc), I opted to handle them both in the same thread.

Furthermore, I would have also preferred to only transmit messages at two points in my program (one to send from , one to receive). However, with the use case of handling node failures, I am instead making calls to socket.sendto() at 4 different points in my program.