z5245663 | **Python v3.7.4** | Link-State Routing Protocol | COMP9331-19T2 Assignment

Terms:

- LSA = Link-State Advertisement.
- NFP = Node-Failure Packet.
- HB = Heartbeat Packet.

LSR Features:

- Link-State broadcast to neighbouring routers every 1 second.
- Restricted broadcast (Only to unvisited neighbours and only for new packet sequences).
- Applies a keepalive mechanism to keep neighbouring nodes alive.
- Fast detection of failed nodes. It takes 0.6 seconds after a node has failed for all other nodes to become aware.
- Handles node failure properly, and accurately updates ROUTE_INTERVAL_UPDATE.
- Handles re-joining nodes properly.
- Handles node isolation properly.
- Implements Dijkstra's algorithm efficiently and recursively calculates least cost path to each node in topology.

After reading config file and initializing node information and its neighbouring nodes, multiple threads will start (All threads are classes and all threads run indefinitely for the life of the node). Each node controls 4 sockets, one of which is bound to the given port in config file and used for listening in listen thread.

Threads Overview:

Thread	Functionality
Heartbeat	broadcast heartbeat packets to active neighbours every 0.2 seconds.
Listen	listen for all incoming packets.
Broadcast	broadcast LSA to active neighbours every 1 second.
ReTransmit	retransmit new LSA + NFP to neighbours from reTransmitQueue.
Dijkstra	calculate least cost path to each node in TOPOLOGY every ROUTE_UPDATE_INTERVAL when no fail occurred, or after 2*ROUTE_UPDATE_INTERVAL if fail occurred.
Neighbour	initialize and store neighbour's data, keep it alive, and detect failure.
Main	Open and read config file, initialize node and neighbours, start threads.

Data Structures:

Node (class): Stores last known packet sequence of the node and its state (1 active, 0 dead).

Nodes (dict): Default dictionary of type Node. Keys: node id, values: Node object.

reTransmitQueue (queue): Stores raw packets that need to be processed and retransmitted.

fNodes (list): Stores failed nodes ids.

Edge (namedtuple): Pair of link cost and predecessor node id.

Edges (dict): Default dictionary of type Edge. Keys: node id, values: Edge object.

Neighbour (class thread): node id, port, link cost, node state, queue(size=5) for heartbeat messages.

Neighbours (dict): Default dictionary of type Neighbour. Keys: node id, values: Neighbour object.

Packets: There are 3 classes of packets:

Packet	Type	Properties
LinkStatePacket	1	type, node id, sequence#, visited nodes list, ***content*** {'node': cost}.
HeartbeatPacket	2	type, node id.
NodeFailurePacket	3	type, node id, visited nodes list, time of failure.

LinkStatePacket.content (dict): Default dictionary of type float. Keys: neighbour node id, values: neighbour node link cost.

TOPOLOGY (dict): Default dictionary of type dict. Keys: node id, values: content (from LSA).

Threads Details:

Broadcast:
Creates an LSA containing information about the node's active neighbours, append node + active neighbours to visited nodes list, increment packet sequence number, pack everything and send to active neighbours, then sleeps for 1 second.

Heartbeat: (Keepalive mechanism)
Creates an HB and send it to active neighbours, then sleep for 0.2 seconds.
Because Broadcast thread sleeps for 1 second, it's better to handle HBs in another thread since it has a lower interval of 0.2 seconds to send HBs.

ReTransmit: (Restricting broadcast)
It's also better to assign a separate thread to process and retransmit packets that need to be routed depending on visited nodes list in the packet.
ReTransmit thread reads packets from reTransmitQueue, checks visited nodes, pack, and send to unvisited active neighbours (Restricting broadcast).

Listen: (Restricting broadcast, Handling node failure, Handling re-joining nodes)
In effect this leaves the Listen thread to handle incoming packets, unpack, drop previously seen packets (Restricting broadcast), put new LSAs and NFPs in reTransmitQueue and update topology, keep neighbours alive by calling the **keepalive()** function if HB is received, kill nodes and retransmit NFPs if received packet is NFP (Handling node failure), and reactivates a dead node by calling the activate() function if an HB or LSA is received from it (Handling re-joining nodes).

Neighbour: (Handling node failure)
Each neighbour has its own thread, it starts upon receiving HB, it sleeps for 0.6 seconds, and after waking tries to dequeue and clear its own queue, if empty then neighbour node is dead and a local function kill() is called which records time of failure(failPoint), sets node state to 0 to indicate as failed, creates 2 NFPs and puts them in retransmit queue, and adds node id to failed nodes list fNodes() (Handling node failure).
In the case when a mutual neighbouring node becomes aware of a dead node first and sends out NFPs, other neighbours (to the same dead node) if received such packets, will set node state to 0 but won't create NFPs, instead will put the received NFPs into reTransmitQueue to be retransmitted, saving computation time, network traffic, and unifying failPoint.

Dijkstra:
Dijkstra thread sleeps 30 seconds, upon waking if failPoint is recorded, it clears it and delays the algorithm for additional time calculated as follows: 60 - (current time – time of failure)
Which totals for 1 minute following the node failure. After waking, it does the same thing again in a loop if another fail has occurred during the delay period until there’s no later failPoint. This ensures the algorithm will always wait 2*ROUTE_UPDATE_INTERVAL following a node failure before executing.
For example, node A starts at 0 (Dijkstra should run at 30), then B fails at 10 (Dijkstra should run at 70), Dijkstra wakes at 30, reads failPoint = 10, delay = 60 - (30 - 10) = 40. Adding the first 30 seconds we get 70. let’s say another node C failed at 40 during the additional delay (Dijkstra should run at 100). Dijkstra wakes at 70, reads failPoint = 40, delay = 60 - (70 – 40) = 30.

<u>Executing:</u> Make a copy of topology, delete failed nodes from it, initialize Edges with node id (cost 0.0, node id), iterate over topology to find the neighbour link with least cost, add it to Edges as starting point (initNode), iterate over initNode neighbours, if the neighbour isn’t in Edges add it with the associated cost and the predecessor as initNode, if it is in Edges then compare and update accordingly, after finishing iterating over initNode neighbours delete initNode from the topology copy. Now enter a for super-loop for nodes in Edges (excluding root node), iterate in a for sub-loop over its neighbours in topology copy, compare and update as before, delete node from topology copy, break and repeat for the length of topology (original).

At the end, for each node in Edges call **path(node):** a recursive function that starts at target node and traces its way back to the root. Base case when reaches link cost 0.0 which is the cost for root node. returns a list of nodes from target to root. Reverse it and print to output.

Potential improvements and other notes:

- Can be easily scaled to include new unknown nodes to join the network by simply creating their threads when receiving HB and adding to Neighbours dict.
- Number of neighbours in config files has no use in the script but kept for future use.
- Since the node knows its neighbours' ports, and won't use other nodes ports for direct communication, then sending its own port won't have any effect because its neighbours also know its port from their config files. Also kept it (commented out) for future use.
- At first I wanted to implement a Timer object but couldn’t figure out a way to manipulate it (resetting/restarting, or add time to it) because a thread can only have one timer and can only start once, therefore I opted for time.sleep() which worked very well.
- Tasks are efficiently distributed among threads, no heavy load on one thread alone.