COMP9331 report

Implementation:

There are six mainly threads for implementing LSR protocol:

Listening thread:

Receiving other's link-state packets and put them into a receive queue in which the main thread will get link-state packets from

Sending thread:

Sending any packet include forward packet, broadcast packet and update packet in the sending queue in which the main thread produce packet and put in.

Main thread:

Processing the data fetched from the receive queue and being responsible for updating the network topology map and constructing the forwarded data packet.

Dijkstra thread:

Calculate the shortest path based on the network topology map.

Check alive thread:

Check if the neighborhood of the node is still online at intervals.

Broadcast thread:

Every second, the link-state packet is generated according to the current latest topology map and placed in the send queue to be sent by the sending thread.

Mutex lock been used for avoiding mess because some variables, such as network topology maps and neighbor heartbeat packet timeouts, are accessed simultaneously by multiple threads.

Features:

1. Correct parsing of configuration files and parameters
2. Broadcast mechanism. Broadcasting link-state packet to neighbors every UPDATE_INTERVAL
3. Forward mechanism. Node will forward the message included in its neighbors' broadcast packet. This means that node will specially note that this message is a forward message.
4. All packet used UDP for sending.
5. Global view network topology. The node generates the latest network topology map based on the received broadcast packets, forwarding packets, and update packets.
6. Least-cost path. Every ROUTER_UPDATE_INTERVAL print the least-cost path calculated by Dijkstra's algorithm.
7. Restricting link-state broadcast. Detail see above (Restricting link-state broadcast).
8. Detecting neighbors failures and delete it from network topology.
9. Detecting neighbors join back again and add it to network topology.

Data structure:

Network topology:

{
source : {
destination : cost
},
source : {
destination : cost,
destination : cost
}
}
	{
	'B' : {
	'F':2.2
	},
	'F' : {
	'B' : 2.2,
	'A' : 4
	}
	}

There is a dictionary whose keys contains all vertex of network topology and the value of each key is a new dictionary whose keys contains all past key's neighbors and the value of each key is the cost from source to destination.

Link-state packet format:

Packet be sent after be converted to json.

SOURCE ID	TYPE	TIMESTAMP	STATE DATA

SOURCE ID:

This is the node ID of producer of link-state data instead of the sender of packet. If A forward B's broadcast to C, the SOURCE ID of packet C received is B not A. This feature will be used for restricting link-state broadcasts.

TYPE:

It's integer selected from 1, 2, 3, 5.

1: broadcast packet. Containing node's link-state information.

2: forward packet. This indicates that this packet is forwarded via other nodes and is not directly reach the receiver.

3: ACK. For acknowledging forwarder that its forward packet has been received.

4: Join-back notice packet. Notice sender's neighbors it joins back and request neighbors to forward others' broadcast to itself.

TIMESTAMP:

The time of generation of information in the packet.

STATE DATA:

Link-state information

Restricting link-state broadcast implement:

There is a dictionary named 'packet_update_time' which record the timestamp of forward packet from one node to another node. For example, A broadcast to B and B forward this link-state information to C then B will record '(A, C): 1234567' in dictionary. (A, C) is key and '1234567' is value. This means B forward a broadcast produced by A to C and the timestamp of broadcast packet is '1234567'.

In the restricts excessive link-state broadcast case, when another node named D also receive the

broadcast from A and forward it to B, B will not forward this forward packet from D to C because B has already record that it has forwarded a link-state packet produced by A to C.

While this mechanism will cause that when a node failure and join back, it will never receive any forward packet. For this reason, we need to construct a new type packet for informing others that 'please **re-forward** any broadcast to me.'

Node failures implement:

There is a dictionary named 'node_heartbeat' whose key is ID of node and value is the lost time of broadcast of corresponding node. Every one second Check alive thread will add 1 to the value of all keys. The value of key is larger than 3 means the node named key failures. While when node receive the broadcast of another node named key, main thread will reset the value of key in node_heartbeat to 0. This ensure that the corresponding node to be deemed failures only if the broadcast packet is lost three consecutive times.

While this mechanism only changes the network topology of neighbor of lost node. If a node and a lost node are indirectly adjacent, it will not detect lost node failures. For this reason, the neighbors of lost has responsibility of broadcasting its neighbor has failure.

In addition, when a node joins back, it will send a type 5 packet. This tell the neighbor of packet sender that 'I join back, please **re-forward** any broadcast to me.'

Design trade-offs considered and made:

Some have already mentioned in past paragraph such as node cannot receive forward packet when it join back and indirectly adjacent node cannot detect each other's failure.

In addition, the lost of forward packet is also a big issue. Node send the same broadcast only one time while in a real network we cannot ensure the receiver could receive the forward via UDP. For this reason, in my program when node receive a forward packet, it will reply a ACK to sender of forward packet. If sender doesn't receive the ACK, it will re-send.

Finally, the deadlock is also a problem when I code Multithreading. There are many code need to manipulate multiple shared variables at the same time. For avoiding deadlock, I sort the order of resource requests. This means assume the order is A, B, C, D, if a thread need to require A, B, C, it need to require A then require B and then require C in order. This means if A has already be required by other thread, if not allowed to require C and D. This will avoid deadlock.

Improve:

Then a node join back again, the cost maybe changed.
The cost in different directions between the two points may be different, this also should be considered.