COMP9331

Assignment

Link State Routing Protocol

Student: Jagadish Raghavan Zid: z5226835

Overview of implementation:

The Algorithm is implemented in python3. The code first reads the config file and creates a broadcast message with information regarding it's neighbours. This broadcast message is then transmitted to it's neighbours continuously every 1 second. A heartbeat message is also transmitted to the node's neighbours once every 600 milliseconds to check for node failure. At the receiving end the node transmits the received message to it's neighbours immediately. Dijkstra Algorithm is used to calculate and display the shortest path information every 30 seconds.

Broadcast and heartbeat message format:

`N*` – neighbour no. *
`NW*` – neighbour weight *
`MN*` – missing node *
`HN` – Host Node
`Seq_no` – Sequence Number

The broadcast message is transmitted in the following fashion.

```
Seq_no HN N1 NW1 N2 NW2 … N(last) NW(last)
```

If a neighbouring node goes missing then the broadcast message is of the form below. Here 'Missing' is a plain text to identify the missing nodes at the receiver end.

```
Seq_no HN N1 NW1 N2 NW2 … N(last) NW(last) Missing MN1 MN2 …
```

The heartbeat message is transmitted as given below. Here 'hb' is just plain text to identify it is a heartbeat signal and not a broadcast signal. This message is transmitted only to the host's neighbours.

```
hb HN
```

Variables:

The important variables used to store useful information and their corresponding datatypes are given below:

- `starting_node` : A string which stores the name of the host node.
- `neighbour` : A dictionary which has the names of the neighbouring nodes as keys and a list containing the weights and the corresponding port number as the value.
- `broadcastMessage` : A string which stores the broadcast message.
- `hbMessage` : A string which stores the heartbeat message.

- `Weight` : A dictionary which has the names of all neighbouring nodes of the received broadcast as keys and their corresponding weights as their values. The dictionary changes each time it receives a new broadcast message.
- `Graph` : A dictionary which has the names of all nodes broadcasting to the host as keys and the corresponding `weight` dictionary as the value.

Algorithm Features:

- **Broadcasting message:**
  The code uses threads to transmit the broadcast and heartbeat messages to it's neighbours in order to carry out multiple transmissions and calculations simultaneously in regular intervals.

- **Excessive Link-State Broadcast Restriction:**
  A sequence number is used to restrict the excessive broadcast. The maximum value of the sequence number is 10000 and will reset to 0 each time it exceeds this value. Every time a new broadcast message is received by a node, it checks if the sequence number for the sending node is appended in a dictionary where the sending node is the key and a list of sequence numbers received from the sender is the value. If a given sequence number for a particular sender is already present in this list, then the message is discarded and would not be sent to the host's neighbours. The list for each sender is cleared each time the sequence number reaches 10000.

- **Node failure check:**
  As mentioned before a heartbeat message is transmitted to the neighbours of the host. A dictionary with each neighbour as the key stores the count of the heartbeat message sent by the corresponding neighbour. This count is checked with a previous count value saved 4 seconds before. If the new count value is the same as the count value 4 seconds before, then that node is considered to be dead. This checking is done every 4 seconds. If the count value reaches 10000 it resets to 0 and the previous value rests to -1 in order to avoid overflow of the counter. The missing node is now removed from the `neighbour` and `graph` dictionaries. The missing node's weight and port number are stored in a `del_group` list which is used later if the missing node rejoins.
  The name of the node deemed dead is now attached at the end of the broadcast message followed by a 'Missing' text after the weights have been sent. The receivers process this and remove the missing nodes from their `graph` dictionary and forward the same message to their neighbours until the entire network knows of the failed node.

- **Node Re-join:**
  If any node which has been disconnected is re-joining the network, it's neighbours are not going to have information about it as it was removed from the `neighbour` dictionary when the node failed. But this information is present in the `del_group` list. Any broadcast message received at a receiver is checked if the sender is present in the `del_group` list. If so then the sender is a reconnecting node and the information regarding the re-joining node is added to the `neighbour` dictionary and the broadcast message is modified accordingly.

- **Dijkstra Algorithm:**
  The weight to each node int the network form the host node is initially set as `1000000` except the host node for which it is set as 0. The node name, it's weight and previous node (initialised to host node) are then inserted into a priority queue where the distance is the key. Now we iteratively remove the minimum value from the priority queue, store it in a list and calculate the distance to the neighbouring nodes of the node with minimum weight from the source node. If the new calculated weight for a node is less than what is present in the priority queue for the same node, the new weight is updated in the priority queue along with the node from which it achieved the smaller weight.
  Once the priority queue is empty we have a list which contains the minimum path length and the previous node to achieve it for all nodes in the network. We canss then backtrack the previous nodes and get the shortest path for any node in the network.

Scope for improvement:

The utilisation of memory can be improved as there are a lot of dictionaries and lists each storing 10000s of pieces of information. Maybe using dynamic memory allocation can help free up this excessive usage of memory.

Sample Outputs: